Stanford Heuristic Programming Project
Memo HPP-79-12

June 1979

Computer Science Department
Report No. STAN-CS-79-742

12

Search

by

Anne Gardner

a section of the

Handbook of Artificial Intelligence

edited by

Avron Barr and Edward A. Feigenbaum

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER HPP-79-12 (STAN-CS-79-742)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) SEARCH. ~~A~~ section of the Handbook of Artificial Intelligence.		5. TYPE OF REPORT & PERIOD COVERED technical, June 1979
		6. PERFORMING ORG. REPORT NUMBER HPP-79-12 (STAN-CS-79-742)
7. AUTHOR(s) (10) Anne Gardner (edited by Avron Barr and Edward A. Feigenbaum)		8. CONTRACT OR GRANT NUMBER(s) (15) MDA 903-77-C-0322, ARPA Order-3423
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Ave., Arlington, VA 22209		12. REPORT DATE (11) June 1979
		13. NUMBER OF PAGES 112
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

(12) 113p.

Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(9) Technical rept.,

18. SUPPLEMENTARY NOTES

(14) STAN-CS-79-742, HPP-79-12

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

094 120

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Those of us involved in the creation of the Handbook of Artificial Intelligence, both writers and editors, have attempted to make the concepts, methods, tools, and main results of artificial intelligence research accessible to a broad scientific and engineering audience. Currently AI work is familiar mainly to its practicing specialists and other interested computer scientists. Yet the field is of growing interdisciplinary interest and practical importance. With this book we are trying to build bridges that are easily crossed by engineers, scientists in other fields, and our own computer science colleagues.

In the Handbook we intend to cover the breadth and depth of AI, presenting general overviews of the scientific issues, as well as detailed discussions of particular techniques and important AI systems. Throughout we have tried to keep in mind the reader who is not a specialist in AI.

As the cost of computation continues to fall, new areas of application of computers become potentially viable. For many of these areas, there do not exist mathematical "cores" to structure calculational use of the computer. Such areas will inevitably be served by symbolic models and symbolic inference techniques. Yet those who understand symbolic computation have been speaking largely to themselves for twenty years. We feel that it is urgent for AI to "go public" in the manner intended by the Handbook.

Several other writers have recognized a need for more widespread knowledge of AI and have attempted to help fill the vacuum. Lay reviews, in particular Margaret Boden's Artificial Intelligence and Natural Man, have tried to explain what is important and interesting about AI, and how research in AI progresses through our programs. In addition, there are a few textbooks that attempt to present a more detailed view of selected areas of AI, for the serious student of computer science. But no textbook can hope to describe all of the sub-areas, present brief explanations of the important ideas and techniques, and review the 40 or 50 most important AI systems.

The Handbook contains several different types of articles. Key AI ideas and techniques are described in core articles (e.g. basic concepts in heuristic search, semantic nets). Important individual AI programs (e.g. SHRDLU) are described in separate articles that indicate among other things the designer's goal, the techniques employed, and the reasons why the program is important. Overview articles discuss the problems and approaches in each major area. The overview articles should be particularly useful to those who seek a summary of the underlying issues that motivate AI research.

Eventually the Handbook will contain approximately two hundred articles. We hope that the appearance of this material will stimulate interaction and cooperation with other AI research sites. We look forward to being advised of our inevitable errors of omission and commission. And for a field as fast moving as AI, it is important that its practitioners alert us to important developments, so that future editions will reflect this new material. We intend that the Handbook of Artificial Intelligence be a living and changing reference work.

The articles in this edition of the Handbook were written primarily by graduate students in AI at Stanford University, with assistance from graduate students and AI professionals at other institutions. We wish particularly to acknowledge the help from those at Rutgers University, SRI International, Xerox Palo Alto Research Center, MIT, and the RAND Corporation.

The author of this report, which contains the section of the Handbook on Search, is Anne Gardner. Others who contributed to or commented on earlier versions of this section include Bruce Buchanan, Lew Creary, Jim Davidson, Nils Nilsson, Ira Pohl, Reid Smith, Mark Stefik, Helen Tognetti, and Dave Wilkins.

Search

by

Anne Gardner

a section of the

Handbook of Artificial Intelligence

edited by

Avron Barr and Edward A. Feigenbaum

This research was supported by both the Defense Advanced Research Projects Agency (ARPA Order No. 3423, Contract No. MDA 903-77-C-0322) and the National Institutes of Health (Contract No. NIH RR-00785-06). The views and conclusions of this document should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency, the National Institutes of Health, or the United States Goverment.

Foreword

Those of us involved in the creation of the Handbook of Artificial Intelligence, both writers and editors, have attempted to make the concepts, methods, tools, and main results of artificial intelligence research accessible to a broad scientific and engineering audience. Currently AI work is familiar mainly to its practicing specialists and other interested computer scientists. Yet the field is of growing interdisciplinary interest and practical importance. With this book we are trying to build bridges that are easily crossed by engineers, scientists in other fields, and our own computer science colleagues.

In the Handbook we intend to cover the breadth and depth of AI, presenting general overviews of the scientific issues, as well as detailed discussions of particular techniques and important AI systems. Throughout we have tried to keep in mind the reader who is not a specialist in AI.

As the cost of computation continues to fall, new areas of application of computers become potentially viable. For many of these areas, there do not exist mathematical "cores" to structure calculational use of the computer. Such areas will inevitably be served by symbolic models and symbolic inference techniques. Yet those who understand symbolic computation have been speaking largely to themselves for twenty years. We feel that it is urgent for AI to "go public" in the manner intended by the Handbook.

Several other writers have recognized a need for more widespread knowledge of AI and have attempted to help fill the vacuum. Lay reviews, in particular Margaret Boden's Artificial Intelligence and Natural Man, have tried to explain what is important and interesting about AI, and how research in AI progresses through our programs. In addition, there are a few textbooks that attempt to present a more detailed view of selected areas of AI, for the serious student of computer science. But no textbook can hope to describe all of the sub-areas, present brief explanations of the important ideas and techniques, and review the 40 or 50 most important AI systems.

The Handbook contains several different types of articles. Key AI ideas and techniques are described in core articles (e.g. basic concepts in heuristic search, semantic nets). Important individual AI programs (e.g. SHRDLU) are described in separate articles that indicate among other things the designer's goal, the techniques employed, and the reasons why the program is important. Overview articles discuss the problems and approaches in each major area. The overview articles should be particularly useful to those who seek a summary of the underlying issues that motivate AI research.

Eventually the Handbook will contain approximately two hundred articles. We hope that the appearance of this material will stimulate interaction and cooperation with other AI research sites. We look forward to being advised of our inevitable errors of omission and commission. And for a field as fast moving as AI, it is important that its practitioners alert us to important developments, so that future editions will reflect this new material. We intend that the Handbook of Artificial Intelligence be a living and changing reference work.

The articles in this edition of the Handbook were written primarily by graduate students in AI at Stanford University, with assistance from graduate students and AI professionals at other institutions. We wish particularly to acknowledge the help from those at Rutgers University, SRI International, Xerox Palo Alto Research Center, MIT, and the RAND Corporation.

The author of this report, which contains the section of the Handbook on Search, is Anne Gardner. Others who contributed to or commented on earlier versions of this section include Bruce Buchanan, Lew Creary, Jim Davidson, Nils Nilsson, Ira Pohl, Reid Smith, Mark Stefik, Helen Tognetti, and Dave Wilkins.

Avron Barr
Edward Feigenbaum

Stanford University
July, 1979

Handbook of Artificial Intelligence

Topic Outline

Volumes I and II

The following sections of the Handbook are still in preparation and will appear in the third volume:

Search

Table of Contents

A. Overview

In Artificial Intelligence the terms *problem solving* and *search* refer to a large body of core ideas that deal with deduction, inference, planning, commonsense reasoning, theorem proving, and related processes. Applications of these general ideas are found in programs for natural language understanding, information retrieval, automatic programming, robotics, scene analysis, game playing, expert systems, and mathematical theorem proving. In this chapter we examine search as a tool for problem solving in a more limited area. Most of the examples to be considered in detail are problems that are relatively easy to formalize. Some typical problems are

-- finding the solution to a puzzle;

-- finding a proof for a theorem in logic or mathematics;

-- finding the shortest path connecting a set of nonequidistant points (the traveling-salesman problem);

-- finding a sequence of moves that will win a game, or the best move to make at a given point in a game; and

-- finding a sequence of transformations that will solve a symbolic integration problem.

Organization of the Chapter

This overview takes a general look at search in problem solving, indicating some connections with topics considered in other chapters. The articles in the next section, Section B, describe the problem representations that form the basis of search techniques. The detailed examples there of state-space and problem-reduction representations will clarify what is meant by words like "search" and "problem solving" in AI. Readers to whom the subject of search is new are encouraged to turn to those articles for more concrete presentations of the fundamental ideas. Section B also discusses game trees, which are a historically and conceptually important class of representations.

Section C, Search Methods, deals with the algorithms that use these various problem representations. *Blind* search algorithms, which treat the search space syntactically, are contrasted with *heuristic* methods, which use information about the nature and structure of the problem domain to limit the search. Various search algorithms are presented in full.

Finally, Section D reviews some well-known early programs based on search. It also describes two programs, STRIPS and ABSTRIPS, that introduce the closely related topic of *planning* in problem solving. This general topic, however, is treated more fully under Planning.

Components of Search Systems

Problem-solving systems can usually be described in terms of three main components. The first of these is a *database*, which describes both the current task-domain situation and the goal. The database can consist of a variety of different kinds of data structures including arrays, lists, sets of predicate calculus expressions, property list structures, and semantic networks. In theorem proving, for example, the current task-domain situation consists of assertions representing axioms, lemmas, and theorems already proved; the goal is an assertion representing the theorem to be proved. In information retrieval applications, the current situation consists of a set of facts, and the goal is the query to be answered. In robot problem solving, a current situation is a *world model* consisting of statements describing the physical surroundings of the robot, and the goal is a description that is to be made true by a sequence of robot actions.

The second component of problem-solving systems is a set of *operators* that are used to manipulate the database. Some examples of operators include:

-- in theorem proving, rules of inference such as modus ponens and resolution;

-- in chess, rules for moving chessmen;

-- in symbolic integration, rules for simplifying the forms to be integrated, such as integration by parts or trigonometric substitution.

Sometimes the set of operators consists of only a few general rules of inference that generate new assertions from existing ones. Usually it is more efficient to use a large number of very specialized operators that generate new assertions only from very specific existing ones.

The third component of a problem-solving system is a *control strategy* for deciding what to do next--in particular, what operator to apply and where to apply it. Sometimes control is highly centralized, in a separate control executive that decides how problem-solving resources should be expended. Sometimes control is diffusely spread among the operators themselves.

Another aspect of control strategy is its effect on the contents and organization of the database. In general, the object is to achieve the goal by applying an appropriate sequence of operators to an initial task-domain situation. Each application of an operator modifies the situation in some way. If several different operator sequences are worth considering, the representation often maintains data structures showing the effects on the task situation of each alternative sequence. Such a representation permits a control strategy that investigates various operator sequences in parallel or that alternates attention among a number of sequences that look relatively promising. This is the character of most of the algorithms considered in this chapter; they assume a database containing descriptions of multiple task-domain situations or *states* (see, e.g., C1, Blind State-space Search). It may be, however, that the description of a task-domain situation is too large for multiple versions to be stored explicitly; in this case, a *backtracking* control strategy may be used (see AI **Programming Languages**). A third possibility, available in some types of problems such as theorem proving, exists where the application of operators can add new assertions to the description of the task-domain situation but never can require the deletion of existing assertions. In this case, the database can describe a single, incrementally changing task-domain situation; multiple or alternative descriptions are unnecessary. (See **Theorem Proving**.)

Reasoning Forward and Reasoning Backward

The application of operators to those structures in the database that describe the task-domain situation--to produce a modified situation--is often called *reasoning forward*. The object is to bring the situation, or problem state, forward from its initial configuration to one satisfying a goal condition. For example, an initial situation might be the placement of chessmen on the board at the beginning of the game; the desired goal, any board configuration that is a checkmate; and the operators, rules for the legal moves in chess.

An alternative strategy, reasoning backward, involves using another type of operator, which is applied not to a current task-domain situation but to the goal. The goal statement, or problem statement, is converted to one or more subgoals that are (one hopes) easier to solve and whose solutions are sufficient to solve the original problem. These subgoals may in turn be reduced to sub-subgoals, and so on, until each of them is either accepted to be a trivial problem or its solution is accomplished by the solution of its subproblems. For example, given an initial goal of integrating $1/(\cos x)^2$ dx, and an operator permitting $1/(\cos x)$ to be rewritten as $(\sec x)$, one can work backward toward a restatement of the goal in a form whose solution is immediate: The integral of $(\sec x)^2$ is $\tan x$.

The former approach is said to use *forward reasoning* and to be *data-driven* or *bottom-up*. The latter uses *backward reasoning* and is *goal-directed* or *top-down*. The distinction between forward and backward reasoning assumes that the current task-domain situation or state is distinct from the goal. If one chooses to say that a current state is the state of having a particular goal, the distinction naturally vanishes.

Much human problem-solving behavior is observed to involve reasoning backward, and many artificial intelligence programs are based on this general strategy. In addition, combinations of forward and backward reasoning are possible. One important AI technique involving forward and backward reasoning is called *means-ends analysis*; it involves comparing the current goal with a current task-domain situation to extract a *difference* between them. This difference is then used to index that (forward) operator most relevant to reducing the difference. If this especially relevant operator cannot be immediately applied to the present problem state, subgoals are set up to change the problem state so that the relevant operator can be applied. After these subgoals are solved, the relevant operator is applied and the resulting, modified situation becomes a new starting point from which to solve for the original goal. (See D2, GPS; and D5, STRIPS.)

State Spaces and Problem Reduction

A problem-solving system that uses forward reasoning and whose operators each work by producing a single new object--a new state--in the database is said to represent problems in a *state-space representation* (see B1).

A distinction may be drawn between two cases of backward reasoning. In one, each application of an operator to a problem yields exactly one new problem, whose size or difficulty is typically slightly less than that of the previous problem. Systems of this kind will also be referred to, in this chapter, as employing state-space representations. Two instances of such representations are presented later in the chapter. One example is the Logic Theorist program (D1); the other is the backward-reasoning part of Pohl's *bidirectional search* (C1 and C3d).

A more complex kind of backward reasoning occurs if applying an operator may divide the problem into a set of subproblems, perhaps each significantly smaller than the original. An example would be an operator changing the problem of integrating $2/(x^2-1)$ dx into the three subproblems of integrating $1/(x-1)$ dx, integrating $-1/(x+1)$ dx, and adding the results. A system using this kind of backward reasoning, distinguished by the fact that its operators can change a single object into a conjunction of objects, will be said to employ a *problem-reduction representation*. The relationship between problem-reduction and state-space representations is examined further at the end of Article B2.

There may or may not be constraints on the order in which the subproblems generated by a problem-reduction system can be solved. Suppose, for example, that the original problem is to integrate $(f(x) + g(x))$ dx. Applying the obvious operator changes it to the new problem consisting of two integrations, $f(x)$ dx and $g(x)$ dx. Depending on the representation, the new problem can be viewed as made up of either (a) two integration subproblems that can be solved in any order, or (b) two integration subproblems plus the third subproblem of summing the integrals. In the latter case, the third task cannot be done until the first two have been completed.

Besides the state-space and problem-reduction representation approaches, other variations on problem representation are possible. One occurs in connection with game-playing problems, which differ from most other problems by virtue of the existence of adversary moves. A game-playing problem must be represented so as to take into account the opponent's possible moves as well as the player's own. The usual representation is a *game tree* (see B3), which shares many features of a problem-reduction representation. Another variation is relevant to theorem-proving systems, many of which use forward reasoning and operators (rules of inference) that act on conjunctions of objects in the database. Although the representations discussed here assume that each operator takes only a single object as input, it is possible to define a *theorem-proving representation* that provides for multiple-input, single-output operators (Kowalski, 1972; see **Theorem Proving**).

Graph Representation

In either a state-space or a problem-reduction representation, achieving the desired goal can be equated with finding an appropriate finite sequence of applications of available operators. While what one is primarily interested in--the goal situation or the sequence that leads to it--may depend on the problem, the term *search* can always be understood, without misleading consequences, as referring to the search for an appropriate operator sequence.

Tree structures are commonly used in implementing control strategies for the search. In a state-space representation, a tree may be used to represent the set of problem states produced by operator applications. In such a representation, the root node of the tree represents the initial problem situation or state. Each of the new states that can be produced from this initial state by the application of just one operator is represented by a *successor node* of the root node. Subsequent operator applications produce successors of these nodes, etc. Each operator application is represented by a directed *arc* of the tree. In general, the states are represented by a *graph* rather than by a tree since there may exist different paths from the root to any given node. Trees are an important special case, however, and it is usually easier to explain their use than that of graphs. (See B1, State-space Representation.)

In addition to these ordinary trees and graphs used for state-space representations, specialized ones called *AND/OR graphs* are used for problem-reduction problem-solving methods. For problems in which the goal can be reduced to sets of subgoals, AND/OR graphs provide a means for keeping track of which subgoals have been attempted and of which combinations of subgoals are sufficient to achieve the original goal (see Article B2).

The Search Space

The problem of producing a state that satisfies a goal condition can now be formulated as the problem of searching a graph to find a node whose associated state description satisfies the goal. Similarly, search based on a problem-reduction representation can be formulated as the search of an AND/OR graph.

It should be noted that there is a distinction between the graph to be searched and the tree or graph that is constructed as the search proceeds. In the latter, nodes and arcs can be represented by explicit data structures; the only nodes included are those for which paths from the initial state have actually been discovered. This explicit graph, which grows as the search proceeds, will be referred to as a *search graph* or *search tree*.

In contrast, the graph to be searched is ordinarily not explicit. It may be thought of as having one node for every state to which there exists a path from the root. It may even be thought of, less commonly, as having one node for every state that can be described, whether or not a path to it exists. The implicit graph will be called the *state space* or, if generalized to cover non-state-space representations such as AND/OR graphs or game trees, the *search space*. Clearly, many problem domains (such as theorem proving) have an infinite search space, and the search space in others, though finite, is unimaginably large. Estimates of search space size may be based on the total number of nodes (however defined) or on other measures. In chess, for example, the number of different complete plays of the average-length game has been estimated at 10^{120} (Shannon, 1950, 1956), although the number of "good" games is much smaller (see Good, 1968). Even for checkers the size of the search space has been estimated at 10^{40} (Samuel, 1963).

Searching now becomes a problem of making just enough of the search space explicit in a search graph to contain a solution of the original goal. If the search space is a general graph, the search graph may be either a subgraph, or a subgraph that is also a tree, or a tree obtained by representing distinct paths to one search space node with duplicate search graph nodes.

Limiting Search

The critical problem of search is the amount of time and space necessary to find a solution. As the chess and checkers estimates suggest, exhaustive search is rarely feasible for nontrivial problems. Examining all sequences of n moves, for example, would require operating in a search space in which the number of nodes grows exponentially with n. Such a phenomenon is called a *combinatorial explosion*.

There are several complementary approaches to reducing the number of nodes that a search must examine. One important way is to recast the problem so as to reduce the size of the search space. A dramatic, if well-known, example is the mutilated chessboard problem:

Suppose two diagonally opposite corner squares are removed from a standard 8 by 8 square chessboard. Can 31 rectangular dominoes, each the size of exactly two squares, be so placed as to cover precisely the remaining board? (Raphael, 1976, p. 31)

If states are defined to be configurations of dominoes on the mutilated board, and an operator has the effect of placing a domino, the search space for this problem is very large. If, however, one observes that every domino placed must cover both a red square and a black one and that the squares removed are both of one color, the answer is immediate. Unfortunately, little theory exists about how to find good problem representations. Some of the sorts of things such a theory would need to take into account are explored by Amarel (1968), who gives a sequence of six representations for a single problem, each reducing the search space size by redefining the states and operators.

A second aspect concerns search efficiency within a given search space. Several graph- and tree-searching methods have been developed, and these play an important role in the control of problem-solving processes. Of special interest are those graph-searching methods that use *heuristic knowledge* from the problem domain to help focus the search. In some types of problems, these *heuristic search* techniques can prevent a combinatorial explosion of possible solutions. Heuristic search is one of the key contributions of AI to efficient problem solving. Various theorems have been proved about the properties of search techniques, both those that do and those that do not use heuristic information. Briefly, it has been shown that certain types of search methods are guaranteed to find optimal solutions (when such exist). Some of these methods, under certain comparisons, have also been shown to find solutions with a minimal amount of search effort. Graph- and tree-searching algorithms, with and without the use of heuristic information, are discussed at length in Section C.

A third approach addresses the question: Given one representation of a search problem, can a problem-solving system be programmed to find a better representation automatically? The question differs from that of the first approach to limiting search in that here it is the program, not the program designer, that is asked to find the improved representation. One start on answering the question was made by the STRIPS program (D5). STRIPS augments its initial set of operators by discovering, generalizing, and remembering *macro-operators*, composed of sequences of primitive operators, as it gains problem-solving experience. Another idea was used in the ABSTRIPS program (D6), which implements the idea of *planning*, in the sense of defining and solving problems in a search space from which unimportant details have been omitted. The details of the solution are filled in (by smaller searches within the more detailed space) only after a satisfactory outline of a solution, or *plan*, has been found. Planning is a major topic itself; for further discussion, see Planning.

The Meaning of "Heuristic" and "Heuristic Search"

Although the term "heuristic" has long been a key word in AI, its meaning has varied both among authors and over time. In general, its usage is illustrated by example better than by definition, and several of the prime examples are included in the programs of Section D. However, a brief review of the ways "heuristic" and "heuristic search" have been used may provide a useful warning against taking any single definition too seriously.

As an adjective, the most frequently quoted dictionary definition for "heuristic" is "serving to discover." As a noun, referring to an obscure branch of philosophy, the word meant the study of the methods and rules of discovery and invention (see Polya, 1957, p. 112).

When the term came into use to describe AI techniques, some writers made a distinction between methods for discovering solutions and algorithms for producing them. Thus Newell, Shaw, and Simon stated in 1957: "A process that *may* solve a given problem, but offers no guarantees of doing so, is called a *heuristic* for that problem" (Newell, Shaw, & Simon, 1963b, p. 114). But this meaning was not universally accepted. Minsky, for example, said in a 1961 paper:

> The adjective "heuristic," as used here and widely in the literature, means *related to improving problem-solving performance*; as a noun it is also used in regard to any method or trick used to improve the efficiency of a problem-solving program. . . . But imperfect methods are not necessarily heuristic, nor vice versa. Hence "heuristic" should not be regarded as opposite to "foolproof"; this has caused some confusion in the literature. (Minsky, 1963, p. 407n.)

These two definitions refer, though vaguely, to two different sets: devices that improve efficiency and devices that are not guaranteed. Feigenbaum and Feldman (1963, p. 6) apparently limit "heuristic" to devices with both properties:

> A *heuristic (heuristic rule, heuristic method)* is a rule of thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions; in fact, they do not guarantee any solution at all; *all that can be said for a useful heuristic is that it offers solutions which are good enough most of the time.*

Even this definition, however, does not always agree with common usage, because it lacks a historical dimension. A device originally introduced as a heuristic in Feigenbaum and Feldman's sense may later be shown to guarantee an optimal solution after all. When this happens, the label "heuristic" may or may not be dropped. It has not been dropped, for example, with respect to the A* algorithm (C3b). Alpha-beta pruning (C5b), on the other hand, is no longer called a heuristic.

It should be noted that the definitions quoted above, ranging in time from 1957 to 1963, refer to heuristic rules, methods, and programs, but they do not use the term "heuristic search." This composite term appears to have been first introduced in 1965 in a paper by Newell and Ernst, "The Search for Generality" (see Newell & Simon, 1972, p. 888). The paper presented a framework for comparing the methods used in problem-solving programs up to that time. The basic framework, there called heuristic search, was the one called *state-space search* in the present chapter. Blind search methods were included within the heuristic search paradigm.

A similar meaning for heuristic search appears in Newell and Simon, 1972 (pp. 91-105). Again no contrast is drawn between heuristic search and blind search; rather, heuristic search is distinguished from a problem-solving method called *generate-and-test*. The difference between the two is that the latter simply generates elements of the search space (i.e., states) and tests each in turn until it finds one satisfying the goal condition;

whereas in heuristic search the order of generation can depend both on information gained in previous tests and on the characteristics of the goal. But the Newell and Simon distinction is not a hard and fast one. By their 1976 Turing Lecture, they seem to have collapsed the two methods into one:

> Heuristic Search. A second law of qualitative structure for AI is that symbol systems solve problems by generating potential solutions and testing them, that is, by searching. (Newell & Simon, 1976, p. 126)

In the present chapter, the meaning attached to "heuristic search" stems not from Newell and Simon but from Nilsson, whose 1971 book provides the most detailed and influential treatment of the subject that has yet appeared. For Nilsson, the distinction between heuristic search and blind search is the important one. Blind search corresponds approximately to the systematic generation and testing of search space elements, but it operates within a formalism that leaves room for additional information about the specific problem domain to be introduced, rather than excluding it by definition. If such information, going beyond that needed merely to formulate a class of problems as search problems, is in fact introduced, it may be possible to restrict search drastically. Whether or not the restriction is foolproof, the search is then called heuristic rather than blind.

References

See Amarel (1968), Feigenbaum & Feldman (1963), Good (1968), Jackson (1974), Kowalski (1972), Minsky (1963), Newell & Ernst (1965), Newell, Shaw, & Simon (1963b), Newell & Simon (1972), Newell & Simon (1976), Nilsson (1971), Polya (1957), Raphael (1976), Samuel (1963), Shannon (1950), Shannon (1956), and Vanderbrug & Minker (1975).

B. Problem Representation

B1. State-space Representation

A state-space representation of a problem uses two kinds of entities: *states*, which are data structures giving "snapshots" of the condition of the problem at each stage of its solution, and *operators*, which are means for transforming the problem from one state to another.

A straightforward example of state-space representation is the simple, well-known puzzle called the 8-puzzle. An 8-puzzle is a square tray containing 8 square tiles of equal size, numbered 1 to 8. The space for the 9th tile is vacant (see Figure 1).

2	1	6
4		8
7	5	3

Figure 1. An 8-puzzle.

A tile may be moved by sliding it vertically or horizontally into the empty square. The problem is to transform one tile configuration, say that of Figure 1, into another given tile configuration, say that of Figure 2.

1	2	3
8		4
7	6	5

Figure 2. A solution configuration of the 8-puzzle.

A state is a particular configuration of tiles; each state might be represented by a 3 x 3 matrix, similar to Figures 1 and 2. The operators, corresponding to possible moves, might be defined with separate operators for each of tiles 1 through 8. However, a more concise definition is made possible by viewing the empty square as the object to be moved and stating the operators in terms of the movements of this square. In this formulation, only four operators are used:

```
"UP"     (move the blank up one square),
"DOWN"   (move the blank down one square),
"LEFT"   (move the blank left one square),
"RIGHT"  (move the blank right one square).
```

An operator may be inapplicable in certain states, as when it would move the blank outside the tray of tiles.

The set of all attainable states of a problem is often called its *state space*. The 8-

puzzle, for example, has a state space of size 9!/2--since there are 9! configurations of the tiles but only half this number can be reached from any given starting configuration. This comes to only 181,440 possible states. For comparison, see the discussion of chess and checkers in the Overview article.

The four operators defined for the 8-puzzle form a set of *partial functions* on the state space: Each operator, if it applies to a given state at all, returns exactly one new state as its result. In more complex problems, however, the operators often contain variables. If, for a particular state and operator, the variables can be instantiated in more than one way, then each instantiation yields one new state, and the operators of the problem, if they are to be considered as defining functions, are more accurately termed *operator schemata*.

The complete specification of a state-space problem has three components. One is a set O of operators or operator schemata. In addition, one must define a set S of one or more *initial states* and find a predicate defining a set G of *goal states*. A state-space problem is then the triple (S, O, G). A *solution* to the problem is a finite sequence of applications of operators that changes an initial state into a goal state.

A state space can be treated as a directed graph whose nodes are states and whose arcs are operators transforming one state to another. For example, if state 1 is a state to which any of three operators can be applied, transforming it to state 2, 3, or 4, then the corresponding graph would be as in Figure 3. Nodes 2, 3, and 4 are called the *successors* of node 1.

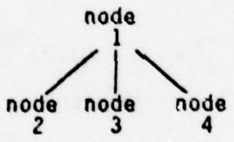


Figure 3. Directed arcs.

In graph notation, a solution to a state-space problem is a path from an initial node to a goal node. In Figure 4, one solution would be an application of operator B twice, followed by operator D, to reach the indicated goal node or final state. There may be other final states and multiple ways to reach a particular final state.

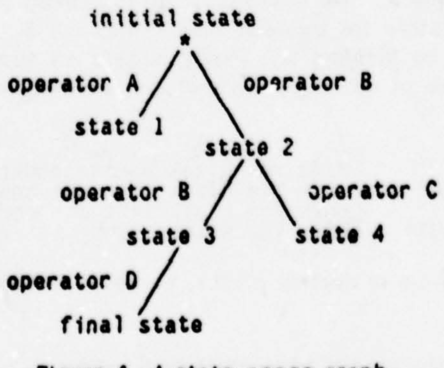
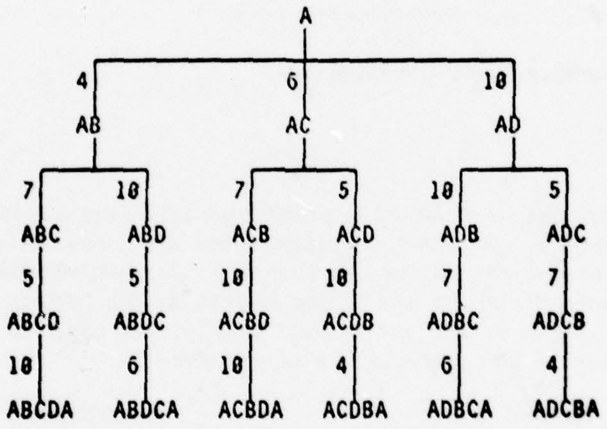


Figure 4. A state-space graph.

A common variation on state-space problems requires finding not just any path but one of minimum cost between an initial node and a goal node. In this case, each arc of the graph is labeled with its cost. An example is the traveling-salesman problem: Given a number of cities to be visited and the mileage between each pair of cities, find a minimum-mileage trip beginning and ending at city A that visits each of the other cities exactly once. An example mileage chart and the corresponding state-space graph are shown in Figure 5. Because different paths to the same city represent distinct partial solutions, each state is identified not just as a city name but as a list of the cities visited so far.

	A	B	C	D
A	–	4	6	10
B		–	7	10
C			–	5
D				–

Mileage chart



State-space graph

Figure 5. A traveling-salesman problem.

The desired solution is A-B-D-C-A, or its reversal, with a total mileage of 25. (The two bottom levels of the graph could be omitted, since the mileage of each tour of *n* cities is determined by the first *n*-1 cities chosen to be visited.)

Because the state-space graph is usually too large to represent explicitly, the problem of searching for a solution becomes one of generating just enough of the graph to contain the desired solution path. Search methods are discussed in Article C1, Blind State-space Search, and Section C3, Heuristic State-space Search.

References

See Nilsson (1971).

B2. Problem-reduction Representation

Often distinguished from the state-space representation of problems is a technique called *problem-reduction representation*. In the problem-reduction approach, the principal data structures are problem descriptions or *goals*. An initial problem description is given; it is solved by a sequence of transformations that ultimately change it into a set of subproblems whose solutions are immediate. The transformations permitted are defined as *operators*. An operator may change a single problem into several subproblems; to solve the former, all the subproblems must be solved. In addition, several different operators may be applicable to a single problem, or the same operator may be applicable in several different ways. In this case it suffices to solve the subproblems produced by any one of the operator applications. A problem whose solution is immediate is called a *primitive problem*. Thus, a problem representation using problem reduction is defined by a triple consisting of

(a) an initial problem description,
(b) a set of operators for transforming problems to subproblems, and
(c) a set of primitive problem descriptions.

Reasoning proceeds backward from the initial goal.

An Example

An example that lends itself nicely to problem-reduction representation is the famous Tower of Hanoi puzzle. In one common version there are three disks, A, B, and C, of graduated sizes. There are also three pegs, 1, 2, and 3. Initially the disks are stacked on peg 1, with A, the smallest, on top and C, the largest, at the bottom. The problem is to transfer the stack to peg 3, as in Figure 1, given that (a) only one disk can be moved at a time, and (b) no disk may be placed on top of a smaller disk.

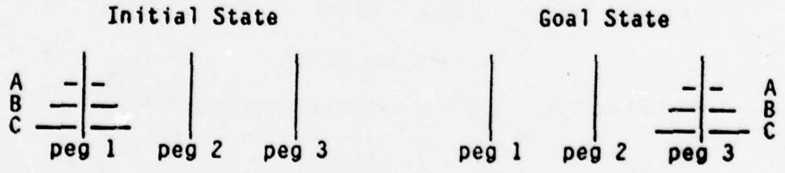


Figure 1. The Tower of Hanoi puzzle.

Only one operator need be used in the solution: Given distinct pegs i, j, and k, the problem of moving a stack of size $n > 1$ from peg i to peg k can be replaced by the three problems:

(a) moving a stack of size $n-1$ from i to j,
(b) moving a stack of size 1 from i to k, and
(c) moving a stack of size $n-1$ from j to k.

The only primitive problem is that of moving a single disk from one peg to another, provided no smaller disk is on the receiving peg. If a smaller disk were present, this problem would be unsolvable (in view of the definition of the only available operator).

Each problem description can now be given by specifying the size *n* of the stack to be moved, the number of the sending peg, and the number of the receiving peg. The original problem, moving a stack of three disks from peg 1 to peg 3, would then be represented as (n = 3, 1 to 3), and the transformation of the original problem to primitive problems can be represented by a tree:

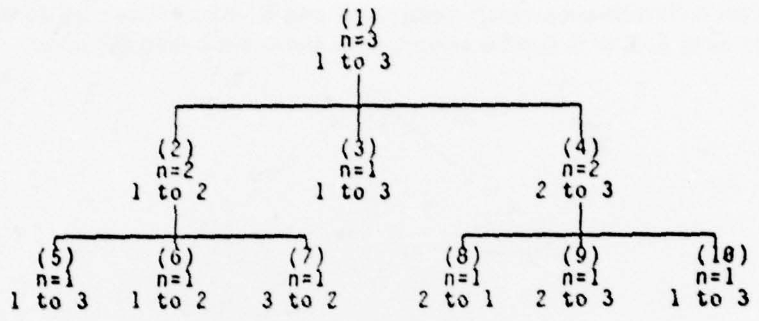


Figure 2. Solution of the Tower of Hanoi puzzle.

There happen to be two possible operator sequences that transform the original problem to primitive problems: Apply the operator to node 1, then node 2, and then node 4; or apply the operator to node 1, then node 4, and then node 2. Since node 3 is a primitive problem, it needs no further attention. Node 2 represents the subproblem of moving the top two disks on peg 1 to peg 2. This subproblem is solved by expanding it to the primitive problems at nodes (5), (6), and (7)--which are solved by moving the smallest disk to peg 3, moving the middle disk to peg 2, and finally putting the small disk back on top of the middle one.

The sequence of operators to be applied should be distinguished from the sequence of actions to be taken to achieve the goal. In the Tower of Hanoi example, the actions are the actual movements of the disks. This sequence is given by the terminal nodes of the tree, read left to right. Whether or not it is considered important to assemble such a sequence of actions depends on the particular problem domain.

AND/OR Graphs

In the example above, a tree was used to display a problem-reduction solution to the Tower of Hanoi puzzle. The tree notation must be generalized if it is to represent the full variety of situations that may occur in problem reduction. This generalized notation for problem reduction is called an *AND/OR graph*.

According to one common formulation (Nilsson, 1971), an AND/OR graph is constructed according to the following rules:

1. Each node represents either a single problem or a set of problems to be solved. The graph contains a start node corresponding to the original problem.

2. A node representing a primitive problem, called a *terminal node*, has no descendants.

3. For each possible application of an operator to problem P, transforming it to a set of subproblems, there is a directed arc from P to a node representing the resulting subproblem set. For example, Figure 3 illustrates the reduction of P to three different subproblem sets: A, B, and C. Since P can be solved if any one of sets A, B, *or* C can be solved, A, B, and C are called *OR nodes*.

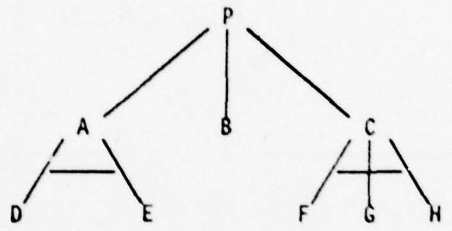


Figure 3. An AND/OR tree.

4. Figure 3 illustrates further the composition of sets A, B, and C: A = {D, E}, B consists of a single (unnamed) problem, and C = {F, G, H}. In general, for each node representing a set of two or more subproblems, there are directed arcs from the node for the set to individual nodes for each subproblem. Since a set of subproblems can be solved only if its members can *all* be solved, the subproblem nodes are called *AND nodes*. To distinguish them from OR nodes, the arcs leading to AND node successors of a common parent are joined by a horizontal line.

5. A simplification of the graph produced by rules 3 and 4 may be made in the special case where only one application of an operator is possible for problem P and where this operator produces a set of more than one subproblem. As Figure 4 illustrates, the *intermediate OR node* representing the subproblem set may then be omitted:

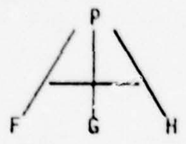


Figure 4. An AND/OR tree with one operator at problem P.

Another example of this construction was given in Figure 2.

In the figures above, every node represents a distinct problem or set of problems. Since each node except the start node has just one parent, the graphs are in fact *AND/OR trees*. As a variation on Figure 3, assume that problem A is reducible to D and E; and problem C, to E, G, and H. Then E may be represented either by two distinct nodes, or by a single node as shown in Figure 5. The choice makes a difference in the search algorithms which are discussed later in the chapter. For example, if node E is in turn reducible to C, the general graph representation simply adds another directed arc to Figure 5, but the corresponding tree becomes infinite.

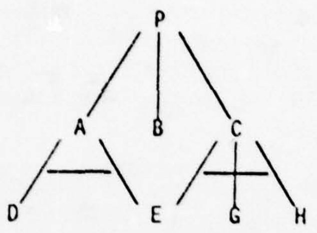


Figure 5. An AND/OR graph.

The constructions discussed so far concern graphs depicting the entire problem search space. To find a solution to the initial problem, one need only build enough of the graph to demonstrate that the start node can be solved. Such a subgraph is called a *solution graph* or, in the more restricted case of an AND/OR tree, a *solution tree*. The following rules apply:

A node is solvable if:

(a) it is a terminal node (a primitive problem);
(b) it is a nonterminal node whose successors are AND nodes that are all solvable; or
(c) it is a nonterminal node whose successors are OR nodes and at least one of them is solvable.

Similarly, a node is unsolvable if:

(a) it is a nonterminal node that has no successors (a nonprimitive problem to which no operator applies);
(b) it is a nonterminal node whose successors are AND nodes and at least one of them is unsolvable; or
(c) it is a nonterminal node whose successors are OR nodes and all of them are unsolvable.

Methods of searching an AND/OR graph for such a solution are discussed in Articles C2 and C4.

Relation between Problem-reduction and State-space Representations

Some interesting general relationships can be found between problem-reduction and state-space representations. In the first place, although one representation often seems the more natural for a given problem, it is often possible to recast the problem definition so that it uses the other form. For example, the Tower of Hanoi puzzle can also be solved by a state-space search using operators that move a single disk and that represent all the legal moves in a given configuration. In comparison to the problem-reduction representation, which in fact gives an algorithm for solving the puzzle, the state-space representation would be a poor one since it leaves room for searching down unnecessarily long paths.

Second, it is possible to translate mechanically between state-space representations and problem-reduction representations without any fundamental shift in the way a problem is viewed. The ways of making such translations can provide helpful insight into many search programs in which the concepts of state-space and problem-reduction representation appear to be intermixed. Several translation schemes are described below. (Some readers may wish to skip the following material at first reading.)

State space to problem reduction. Two approaches suggest themselves for translating state-space representations to problem-reduction representations. In one, the state-space graph is understood as an AND/OR graph containing only OR nodes. Each state of the state-space version corresponds to the problem of getting from that state to a goal state; and a goal state of the state space becomes the primitive problem of getting from that goal state to itself. In other words, data structures representing states are simply reinterpreted as representing problem descriptions, where a problem consists of state information together with an implicit goal.

Alternately, there is a slight variation of the first approach that requires redefining the operators of the state-space representation. Each such operator, taking state i to state j, becomes an operator applicable to the problem of getting from state i to a goal state. Its effect is to reduce the problem to a pair of subproblems: (a) go from state i to state j (a primitive problem), and (b) go from state j to a goal state. Figure 6 illustrates this correspondence.

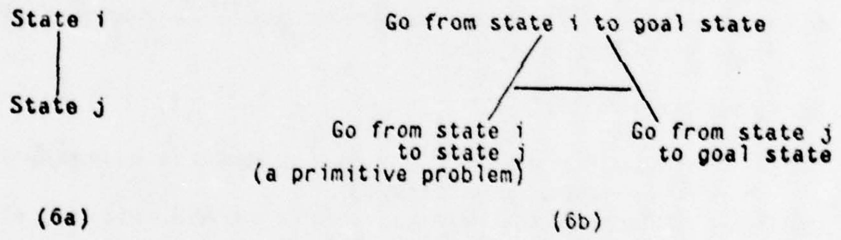

State i
State j
(6a)
Go from state i to goal state
Go from state i to state j (a primitive problem)
Go from state j to goal state
(6b)

Figure 6. (a) Part of a state-space tree; (b) the corresponding part of an AND/OR (problem-reduction) tree.

Problem reduction to state space. The translation from a problem-reduction representation to a state-space representation is a little more complex, assuming that the problem-reduction operators in fact produce AND nodes. The initial problem of the problem-reduction representation can be understood as having two components: (a) the description of the goal to be achieved, as discussed at the beginning of this article, and (b) the description of an initial state of the world. These components will be denoted *g0* and *s0*, respectively. Some examples are

-- *g0* = a theorem to be proved, and *s0* = the axioms from which to prove it;

-- *g0* = a configuration of objects to be achieved, and *s0* = their existing configuration.

Each state S of the corresponding state-space representation is a pair consisting of a stack of goals (gi, ..., g0) to be achieved and a current state s of the world. Thus, the initial state S0 of the state-space representation is S0 = ((g0), s0). A final state is one in which the stack of goals to be achieved has been emptied.

For each problem-reduction operator, mapping a problem or goal g to a set of subgoals {gm, ..., gn}, the state-space representation has a corresponding operator mapping state S1, where S1 = ((gi, ..., g0), s), to a state S2 in which {gm, ..., gn} have been added to the top of the goal-stack (in the order in which they should be carried out, if relevant), and the state of the world s is unchanged; that is, S2 = ((gm, ..., gn, gi, ..., g0), s).

The state-space representation also needs a second type of operator, which becomes applicable whenever the goal on top of the stack represents a primitive problem. Its function is to remove that primitive problem from the stack and, at the same time, to change the state s to reflect its solution. In the Tower of Hanoi puzzle, for example, the new state would reflect the changed position of a single disk. In a theorem-proving problem, the new state would differ from the old one by the addition of one formula to those that had been given as axioms or established from having solved previous subproblems. A representation of this type is used explicitly in Fikes and Nilsson's STRIPS program, described in Article D5.

References

See Jackson (1974), and Nilsson (1971).

B3. Game Trees

Most games played by computer programs, including checkers, chess, *go*, and tic-tac-toe, have several basic features in common. There are two players who alternate in making moves. At each turn, the rules define both what moves are legal and the effect that each possible move will have; there is no element of chance. In contrast to card games in which the players' hands are hidden, each player has complete information about his opponent's position, including the choices open to him and the moves he has made. The game begins from a specified state, often a configuration of men on a board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete *game tree* is a representation of all possible plays of such a game. The root node is the initial state, in which it is the first player's turn to move. Its successors are the states he can reach in one move; their successors are the states resulting from the other player's possible replies; and so on. Terminal states are those representing a win, loss, or draw. Each path from the root node to a terminal node gives a different complete play of the game.

An important difference between a game tree and a state-space tree (Article B1) is that the game tree represents moves of two opposing players, say A and B. An AND/OR tree (Article B2), however, is sufficient to reflect this opposition. The game tree is ordinarily drawn to represent only one player's point of view. In a game tree drawn from A's standpoint, A's possible moves from a given position are represented by OR nodes since they are alternatives under his own control. The moves that B might make in return are AND nodes, since they represent sets of moves to which A must be able to respond. Because the players take turns, OR nodes and AND nodes appear at alternate levels of the tree. In the language of AND/OR graphs, the tree displays the search space for the problem of showing that A can win. A node representing a win for A corresponds to a primitive problem; a node representing a win for B or a draw, to an unsolvable problem. Unlike the usual AND/OR graph terminology, both of these kinds of nodes will be called *terminal nodes*.

As an example, Figure 1 shows a portion of the game tree for tic-tac-toe. The players are X and O, X has the first move, and the tree is drawn from X's standpoint. Positions are considered identical if one can be obtained from the other by rotation or reflection of the grid. The tree could also be drawn from O's standpoint, even though X has the first move. In this case, the AND nodes would become OR nodes, and vice versa, and the labels "win" and "lose" would be reversed. An alternate formulation of game trees, not explicitly distinguishing between AND and OR nodes, is given in Article C5a, Minimax.

Methods of searching a game tree for a winning strategy are discussed in Section C5. As with search in other domains, the source of difficulty in challenging games is the unimaginably large search space. A complete game tree for checkers, for instance, which is harder than tic-tac-toe but far simpler than chess or *go*, has been estimated as having about 10^{40} nonterminal nodes (Samuel, 1963). If one assumed that these nodes could be generated at the rate of 3 billion per second, generation of the whole tree would still require around 10^{21} centuries!

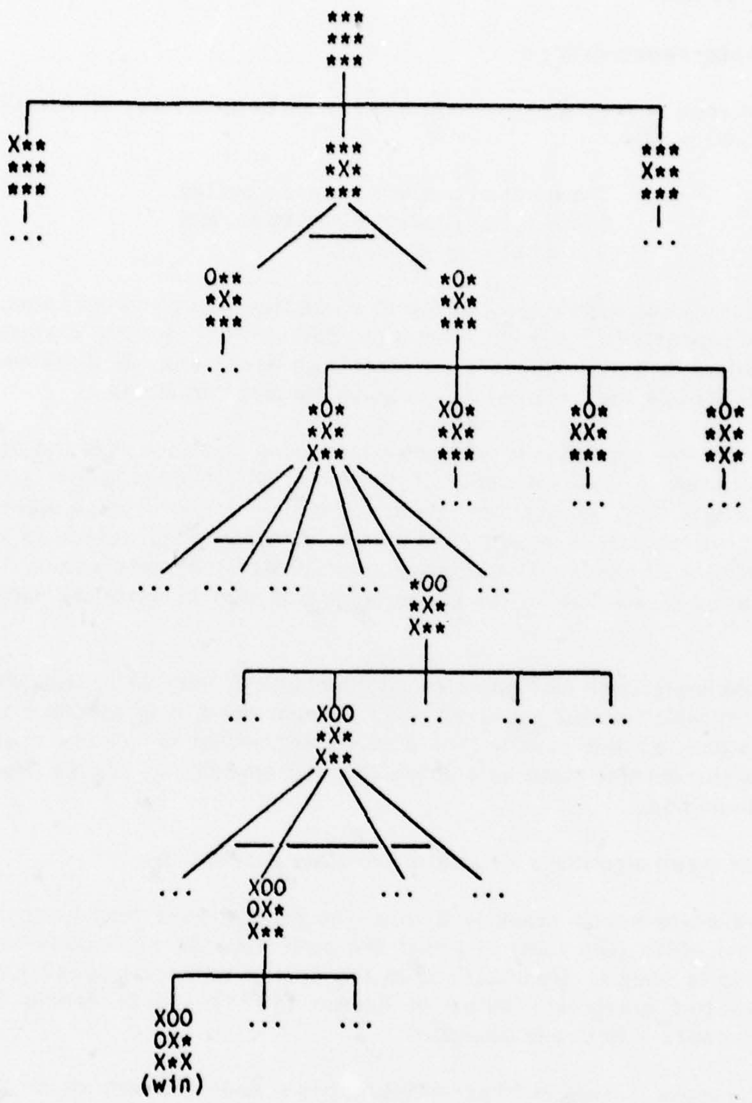


Figure 1. A game tree for Tic-tac-toe.

References

See Nilsson (1971), and Samuel (1963).

C. Search Methods

C1. Blind State-space Search

As discussed in Article B1, a problem in the state-space search paradigm is defined by a triple (S, O, G), where

S is a set of one or more initial states,
O is a set of operators on states, and
G is a set of goal states.

The state space is commonly identified with a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state. A solution is a path from a start state to a goal state. Goal states may be defined either explicitly or as the set of states satisfying a given predicate.

The search for a solution is conducted by making just enough of the state-space graph explicit to contain a solution path. If the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie, the search is called *blind search*. Although blind search is impracticable for nontrivial problems, because of the large proportion of the state space it may explore, it provides a useful foundation for the understanding of *heuristic search* techniques, discussed in Section C3.

Several blind-search methods are described below; they differ from one another mainly in the order in which nodes are examined. In each case, it is assumed that a procedure exists for finding all the *successors* of a given node--that is, all the states that can be reached from the current state by a single operator application. Such a procedure is said to *expand* the given node.

The first three algorithms also make two other assumptions:

(a) The state-space graph is a tree. The implication is that there is only one start state (the root) and that the path from the start node to any other node is unique. Modifications to the search methods needed for a general directed graph are noted in Nilsson (1971) and in Article C3a, Basic Concepts in Heuristic Search.

(b) Whenever a node is expanded, creating a node for each of its successors, the successor nodes contain pointers back to the parent node. When a goal node is finally generated, this feature makes it possible to trace the solution path.

Breadth-first Search

The breadth-first method expands nodes in order of their proximity to the start node, measured by the number of arcs between them. In other words, it considers every possible operator sequence of length n before any sequence of length $n+1$. Thus, although the search may be an extremely long one, it is guaranteed eventually to find the shortest possible solution sequence if any solution exists.

Breadth-first search is described by the following algorithm:

(1) Put the start node on a list, called OPEN, of unexpanded nodes. If the start node is a goal node, the solution has been found.
(2) If OPEN is empty, no solution exists.
(3) Remove the first node, n, from OPEN and place it in a list, called CLOSED, of expanded nodes.
(4) Expand node n. If it has no successors, go to (2).
(5) Place all successors of node n at the end of the OPEN list.
(6) If any of the successors of node n is a goal node, a solution has been found. Otherwise, go to (2).

As an example of breadth-first search, consider a world consisting of a table and three toy blocks. The initial state of the world is that blocks 2 and 3 are on the table, and block 1 is on top of block 2 (see Figure 1). We wish to reach a goal state in which the three blocks are stacked with block 1 on top, block 2 in the middle, and block 3 on the bottom.

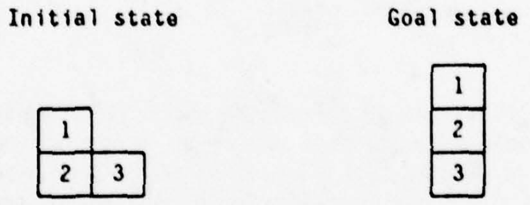


Figure 1. An example problem for breadth-first search.

The only operator is MOVE X to Y, which moves object X onto another object, Y. As *preconditions to applying the operator*, it is required (a) that X, the object to be moved, be a block with nothing on top of it, and (b) that if Y is a block, there must be nothing on Y. Finally, the operator is not to be used to generate the same state more than once. (This last condition can be checked from the lists of expanded and unexpanded nodes.)

Figure 2 shows the search tree generated by the breadth-first algorithm. The nodes are states S0 through S10; node S1, for example, corresponds to the successor state of S0 reached by "MOVE block 1 to the table." The nodes are generated and expanded in the order given by their state numbers, i. e., S0, S1, S2, ... , S10. When the algorithm terminates, finding S10 to be the goal state, the list of expanded nodes contains S0 through S5, and the OPEN list still contains S6 through S10.

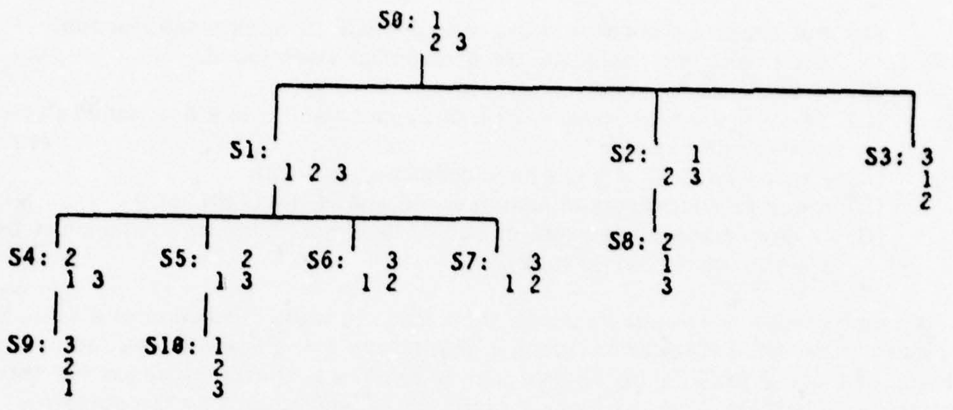


Figure 2. The search tree for Figure 1.

Uniform-cost Search

The breadth-first algorithm can be generalized slightly to solve the problem of finding the cheapest path from the start state to a goal state. A nonnegative cost is associated with every arc joining two nodes; the cost of a solution path is then the sum of the arc costs along the path. The generalized algorithm is called a *uniform-cost search.* If all arcs have equal cost, the algorithm reduces to breadth-first search. The need for assigning costs to the arcs is illustrated by the traveling-salesman problem, described in Article B1, where the different distances between cities correspond to the arc costs and the problem is to minimize the total distance traveled.

In the uniform-cost algorithm given below, the cost of the arc from node i to node j is denoted by c(i,j). The cost of a path from the start node to any node i is denoted g(i).

```
(1) Put the start node, s, on a list called OPEN of unexpanded nodes. If the
    start node is a goal node, a solution has been found. Otherwise, set
    g(s) = 0.
(2) If OPEN is empty, no solution exists.
(3) Select from OPEN a node i such that g(i) is minimum. If several nodes
    qualify, choose node i to be a goal node if there is one; otherwise, choose
    among them arbitrarily. Move node i from OPEN to a list, CLOSED, of
    expanded nodes.
(4) If node i is a goal node, the solution has been found.
(5) Expand node i. If it has no successors, go to (2).
(6) For each successor node j of node i, compute g(j) = g(i) + c(i,j) and place
    all the successor nodes j in OPEN.
(7) Go to (2).
```

Depth-first Search

Depth-first search is characterized by the expansion of the most recently generated, or deepest, node first. Formally, the *depth* of a node in a tree is defined as follows:

> The depth of the start node is 0.
> The depth of any other node is one more than the depth of its predecessor.

As a consequence of expanding the deepest node first, the search follows a single path through the state space downward from the start node; only if it reaches a state that has no successors does it consider an alternate path. Alternate paths systematically vary those previously tried, changing only the last n steps while keeping n as small as possible.

In many problems, of course, the state-space tree may be of infinite depth, or at least may be deeper than some known upper bound on the length of an acceptable solution sequence. To prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors. It should be noted that, even if such a *depth bound* is used, the solution path found is not necessarily the shortest one.

The following algorithm describes depth-first search with a depth bound:

(1) Put the start node on a list, OPEN, of unexpanded nodes. If it is a goal node, a solution has been found.
(2) If OPEN is empty, no solution exists.
(3) Move the first node, n, on OPEN to a list CLOSED of expanded nodes.
(4) If the depth of node n is equal to the maximum depth, go to (2).
(5) Expand node n. If it has no successors, go to (2).
(6) Place all successors of node n at the beginning of OPEN.
(7) If any of the successors of node n is a goal node, a solution has been found. Otherwise go to (2).

As an example, consider the following simple problem: A pawn is required to move through the matrix in Figure 3 from top to bottom. The pawn may enter the matrix anywhere in the top row. From a square containing 0, the pawn must move downward if the square below contains 0; otherwise, it must move horizontally. From a square containing 1, no further moves are possible. The goal is to reach a square containing zero in the bottom row. A depth bound of 5 is assumed.

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & 1 & 0 & 0 & 0 \\ \hline
2 & 0 & 0 & 1 & 0 \\ \hline
3 & 0 & 1 & 0 & 0 \\ \hline
4 & 1 & 0 & 0 & 0 \\ \hline
\end{array}$$

Figure 3. An example problem for depth-first search.

The search tree generated by the depth-first algorithm is shown in Figure 4. At node S0, the pawn has not yet entered the grid. At the other nodes, its position is given as a (row number, column number) pair. The numbering of nodes gives the order in which they are moved out of the OPEN list of unexpanded nodes. When the algorithm terminates, the OPEN list contains S17 (a goal node) and S18; all other nodes are on the expanded list. The solution found, which is one move longer than the minimum, calls for the pawn to enter at (1,3), move one square right, and then go straight down to (4,4). Had no depth bound been used, the tree would have been one level deeper since node S12 has a successor, (4,1). Since the algorithm treats the state space as a tree, not a general graph, it does not discover that the distinct nodes S2 and S9 in fact represent the same state. Consequently, the search downward from S9 duplicates the work already done from S2.

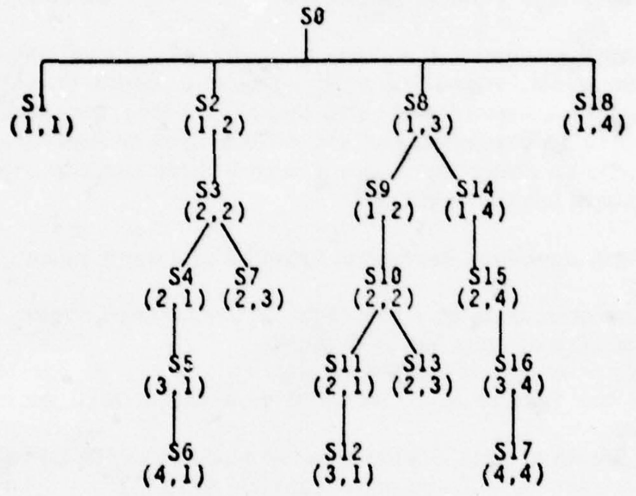


Figure 4. The search tree for Figure 3.

Bidirectional Search

Each of the algorithms given above uses *forward reasoning*, working from the start node of a state-space tree towards a goal node and using operators that each map a node i to a successor node j. In some cases, the search could equally well use *backward reasoning*, moving from the goal state to the start state. An example of this is the 8-puzzle, in which

(a) the goal state can be fully described in advance, and
(b) it is easy to define inverse operators--each applicable operator mapping node j to a predecessor node i.

Since backward search through a tree is trivial, it is assumed that node j can have more than one predecessor--that is, several inverse operators may apply at node j. For example, in the pawn maze problem, Figure 4, position (1,2) [at nodes S2 and S9] would have both nodes S0 and S8 as predecessors.

Forward and backward reasoning can be combined into a technique called *bidirectional* *search*. The idea is to replace a single search graph, which is likely to grow exponentially, by two smaller graphs: one starting from the initial state and one starting from the goal. The search terminates (roughly) when the two graphs intersect.

A bidirectional version of the uniform-cost algorithm, guaranteed to find the shortest solution path through a general state-space graph, is due to Pohl (1969, 1971). Empirical data for randomly generated graphs showed that Pohl's algorithm expanded only about one-fourth as many nodes as unidirectional search.

An algorithm for blind bidirectional search is given in detail below. A related algorithm for *heuristic* bidirectional search is discussed in Article C3d.

The following notation is used in the algorithm:

The start node is s; the goal or terminal node, t.

S-OPEN and S-CLOSED are lists of unexpanded and expanded nodes, respectively, generated from the start node.

T-OPEN and T-CLOSED are lists of unexpanded and expanded nodes, respectively, generated from the terminal node.

The cost associated with the arc from node n to node x is denoted c(n,x).

For a node x generated from the start node, gs(x) measures the shortest path found so far from s to x.

For a node x generated from the terminal node, gt(x) measures the shortest path found so far from x to t.

The algorithm is as follows:

(1) Put s in S-CLOSED, with gs(s) = 0. Expand node s, creating a node for each of its successors. For each successor node x, place x on S-OPEN, attach a pointer back to s, and set gs(x) to c(s,x). Correspondingly, put t in T-CLOSED, with gt(t) = 0. Expand node t, creating a node for each of its predecessors. For each predecessor node x, place x on T-OPEN, attach a pointer forward to t, and set gt(x) = c(x,t).

(2) Decide whether to go forward or backward. If forward, go to (3); if backward, to (4). (One way to implement this step is to alternate between forward and backward moves. Another way, which Pohl found to give better performance, is to move backward if T-OPEN contains fewer nodes than S-OPEN; otherwise, forward. It is assumed that a solution path does exist, so the chosen list will be nonempty.)

(3) Select from S-OPEN a node n at which gs(n) is minimum. Move n to S-CLOSED. If n is also in T-CLOSED, go to (5). Otherwise, for each successor x of n :

(a) If x is on neither S-OPEN nor S-CLOSED, then add it to S-OPEN. Attach a pointer back to n and the path cost gs(x) = gs(n) + c(n,x).

(b) If x was already on S-OPEN, a shorter path to x may have just been found. Compare the previous path cost, gs(x), with the new cost gs(n) + c(n,x). If the latter is smaller, set gs(x) to the new path cost and point x back to n instead of its predecessor on the longer path.

(c) If x was already on S-CLOSED, do nothing; although a new path to x has been found, its cost must be at least as great as the cost of the path already known. (For further consideration of this point, see Article C3b.)

Return to (2).

(4) Select from T-OPEN a node n at which gt(n) is minimum. Move n to T-CLOSED. If n is also in S-CLOSED, go to (5). Otherwise, for each predecessor x of n:

(a) If x is on neither T-OPEN nor T-CLOSED, then add it to T-OPEN. Attach a pointer forward to n and the path cost gt(x) = gt(n) + c(x,n).

(b) If x was already on T-OPEN and a shorter path from x to t has just been found, reduce the stored value of gt(x), and point x forward to n (instead of to its successor on the longer path).

(c) If x was already on T-CLOSED, do nothing.

Return to (2).

(5) Consider the set of nodes that are in both S-CLOSED and either T-CLOSED or T-OPEN. Select from this set a node n for which gs(n) + gt(n) is minimum; and exit with the solution path obtained by tracing the path from n back to s and forward to t.

References

See Nilsson (1971), Pohl (1969), and Pchl (1971).

C2. Blind AND/OR Graph Search

A problem to be solved using AND/OR-graph search can be defined by specifying a start node (representing an initial goal or problem description), a set of terminal nodes (descriptions of primitive problems), and a set of operators for reducing goals to subgoals. The rules for constructing an AND/OR graph, together with the use of such graphs for problem-reduction representation, were discussed in Article B2. To recapitulate briefly, each possible application of an operator at a node n (see Figure 1) is represented by a directed arc from node n to a successor node; these successor nodes are called *OR nodes*, since only one of the operator applications will ever be needed to solve the problem that node n represents. Each OR node successor of node n represents a set of subproblems. If the set of subproblems represented by an OR node m has more than one element, then there are directed arcs from m to nodes representing the individual elements of the set. These successors are called *AND nodes*, because all of the elements of the set must be solved in order to solve the subproblem set represented by node m. To distinguish AND nodes visually from OR nodes, the arcs in the graph from m to its AND successors are joined by a horizontal line.

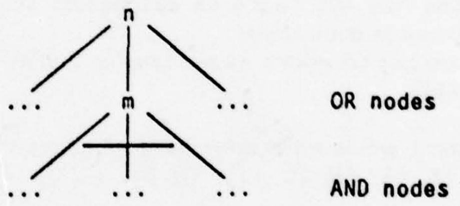


Figure 1. AND/OR graph notation.

Formally, a node or problem is said to be *solved* if one of the following conditions holds:

1. The node is in the set of terminal nodes (primitive problems). (In this case, the node has no successors.)
2. The node has AND nodes as successors and all these successors are solved.
3. The node has OR nodes as successors and any one of these successors is solved.

A solution to the original problem is given by a subgraph of the AND/OR graph sufficient to show that the start node is solved. In Figure 2, for example, assuming that nodes 5, 6, 8, 9, 10, and 11 are all terminal, there are three possible solution subgraphs: {1, 2, 4, 8, 9}, {1, 3, 5, 6, 7, 10}, and {1, 3, 5, 6, 7, 11}.

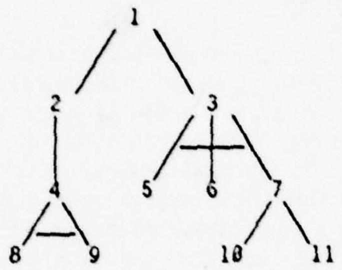


Figure 2. An AND/OR graph.

A node is said to be *unsolvable* if one of the following conditions is true:

1. The node has no successors and is not in the set of terminal nodes. That is, it is a nonprimitive problem to which no operator can be applied.
2. The node has AND nodes as successors and one or more of these successors is unsolvable.
3. The node has OR nodes as successors and all of these succesors are unsolvable.

Again in Figure 2, node 1 would be unsolvable if all nodes in any of the following sets were unsolvable: {8, 5}, {8, 6}, {8, 10, 11}, {9, 5}, {9, 6}, {9, 10, 11}.

Two algorithms for the blind search of an AND/OR tree (breadth-first and depth-first) are given at the end of this article. They have several features in common with blind state-space search algorithms (Article C1): The operation of *expanding* a node is again present, and again the algorithms differ mainly in the order in which nodes are considered for expansion. It should be noted that the expansion of a node may differ slightly from the case of state-space search. In Figure 2, for example, two operators apply at node 1: One reduces it to a single equivalent problem (node 2) and the other to a set (node 3) of three subproblems (nodes 5, 6, and 7). In this case, nodes 2, 3, 5, 6, and 7 would all be generated in expanding node 1, and each new node would be given a pointer to its immediate predecessor, but only nodes 2, 5, 6, and 7 would be placed on the list of unexpanded nodes.

In contrast to the state-space search algorithms, most of which use forward reasoning, the search algorithms below reason backward from the initial goal. The algorithms described here make two important simplifying assumptions: (a) The search space is an AND/OR tree and not a general graph, and (b) when a problem is transformed to a set of subproblems, the subproblems may be solved in any order. The first assumption implies that identical subproblems may arise at different nodes of the search tree and will need to be solved anew whenever one of them is encountered. Modifications needed for searching a general AND/OR graph are discussed in Nilsson (1971). A way of eliminating the second assumption, that all subproblems are independent, is discussed in Article C4, Heuristic Search of an AND/OR Graph.

Breadth-first Search of an AND/OR Tree

The following algorithm describes the breadth-first search of an AND/OR tree. If a solution tree exists, this algorithm finds a solution tree of minimum depth, provided that intermediate OR nodes are ignored in calculating the depth of the tree. The start node is assumed not to be a terminal node.

```
(1) Put the start node on a list, OPEN, of unexpanded nodes.
(2) Remove the first node, n, from OPEN.
(3) Expand node n--generating all its immediate successors and, for each
    successor m, if m represents a set of more than one subproblem, generating
    successors of m corresponding to the individual subproblems. Attach, to
    each newly generated node, a pointer back to its immediate predecessor.
    Place all the new nodes that do not yet have descendants at the end of
    OPEN.
(4) If no successors were generated in (3), then
    (a) Label node n unsolvable.
    (b) If the unsolvability of n makes any of its ancestors unsolvable, label
        these ancestors unsolvable.
    (c) If the start node is labeled unsolvable, exit with failure.
    (d) Remove from OPEN any nodes with an unsolvable ancestor.
(5) Otherwise, if any terminal nodes were generated in (3), then
    (a) Label these terminal nodes solved.
    (b) If the solution of these terminal nodes makes any of their ancestors
        solved, label these ancestors solved.
    (c) If the start node is labeled solved, exit with success.
    (d) Remove from OPEN any nodes that are labeled solved or that have a
        solved ancestor.
(6) Go to step 2.
```

Depth-first Search of an AND/OR Tree

A bounded depth-first search can be obtained by changing only step 3 of the breadth-first algorithm. The revised step 3 is as follows:

```
(3') If the depth of n is less than the depth bound, then: Expand node n, generating
     all its immediate successors and, for each successor m, if m represents a
     set of more than one subproblem, generating successors of m corresponding
     to the individual subproblems. Attach, to each newly generated node, a
     pointer back to its immediate predecessor. Place all the new nodes that do
     not yet have descendants at the beginning of OPEN.
```

The depth-first search will find a solution tree, provided one exists within the depth bound. As with breadth-first search, the notion of depth is more meaningful if intermediate OR nodes are not counted. For this purpose one might add the following to the end of step 3':

For each node x added to OPEN, set the depth of x to be the depth of node n, plus 1.

Given that the start node has depth 0, the depth of any node x will then be the length of the operator sequence that must be applied to reach node x from the start node.

References

See Nilsson (1971).

C3. Heuristic State-space Search

C3a. Basic Concepts in Heuristic Search

In the blind search of a state-space (Article C1) or an AND/OR graph (Article C2), the number of nodes expanded before reaching a solution is likely to be prohibitively large. Because the order of expanding the nodes is purely arbitrary and does not use any properties of the problem being solved, one usually runs out of space or time (or both) in any but the simplest problems. This result is a manifestation of the *combinatorial explosion*.

Information about the particular problem domain can often be brought to bear to help reduce the search. In this section, it is assumed that the definitions of initial states, operators, and goal states all are fixed, thus determining a search space; the question, then, is how to search the given space efficiently. The techniques for doing so usually require additional information about the properties of the specific problem domain beyond that which is built into the state and operator definitions. Information of this sort will be called *heuristic information*, and a search method using it (whether or not the method is foolproof) will be called a *heuristic search method* (Nilsson, 1971).

The Importance of Heuristic Search Theory

Heuristic search methods were employed by nearly all early problem-solving programs. Most of these programs, though, were written to solve problems from a single domain, and the domain-specific information they used was closely intertwined with the techniques for using it. Thus the heuristic techniques themselves were not easily accessible for study and adaptation to new problems, and there was some likelihood that substantially similar techniques would have to be reinvented repeatedly. Consequently, an interest arose in developing generalized heuristic search algorithms, whose properties could be studied independently of the particular programs that might use them. (See Newell & Ernst, 1965; Feigenbaum, 1969; Sandewall, 1971.) This task, in turn, required a way of describing problems that generalized across many different domains. Such generalized problem formulations have been discussed in Section B, Problem Representation, in an approach generally following Nilsson (1971). Given a generalized problem representation, the most basic heuristic search techniques can be studied as variations on blind search methods for the same type of problem representation.

The current state of heuristic search theory has been diversely judged. One of the best known students of the subject has remarked, "The problem of efficiently searching a graph has essentially been solved and thus no longer occupies AI researchers" (Nilsson, 1974). Other work makes it clear, however, that the theory is far from complete (e.g., Gaschnig, 1977; Simon & Kadane, 1975). Its kinship with complexity theory now tends to be emphasized (see Pohl, 1977).

Ways of Using Heuristic Information

The points at which heuristic information can be applied in a search include

(a) deciding which node to expand next, instead of doing the expansions in a strictly breadth-first or depth-first order;

(b) in the course of expanding a node, deciding which successor or successors to generate--instead of blindly generating all possible successors at one time; and

(c) deciding that certain nodes should be discarded, or *pruned*, from the search tree.

A state-space search algorithm is presented below that uses heuristic information only at the first of these points, deciding which node to expand next, on the assumption that nodes are to be expanded fully or not at all. The general idea is always to expand the node that seems "most promising." A search that implements this idea is called an *ordered search* or *best-first search*. Ordered search has been the subject of considerable theoretical study, and several variations on the basic algorithm below are reviewed in articles IIC3b through IIC3d (ordered state-space search) and article IIC4 (ordered AND/OR graph search).

The other two uses of heuristic information can be discussed more briefly. Decisions of the second kind--determining which successors to generate--are often decisions of operator selection, determining which operator to apply next to a given node. A node to which some but not all applicable operators have been applied is said to have been *partially developed* or *partially expanded*. The use of heuristic information to develop nodes partially, reserving the possibility of fuller expansion at a later point in the search, has been investigated by Michie (1967) and by Michie and Ross (1970). Other applications of the idea of limiting the successors of a given node occur in game-playing programs (see C5c). Another important variant of the idea is *means-ends analysis*, which, instead of deciding on an applicable operator, chooses an operator most likely to advance the search whether or not it is immediately applicable. The problem of making the operator applicable, if necessary, is addressed secondarily. (See D2, GPS; and D5, STRIPS.)

The third use of heuristic information, for *pruning*, amounts to deciding that some nodes should never be expanded. In some cases, it can be definitely determined that a node is not part of a solution, and the node may then be safely discarded, or pruned, from the search tree. In other cases pruning may be desirable even though the nodes pruned cannot be guaranteed inessential to a solution. One reason, in conjunction with a best-first search, is simply to save the space that would be required to retain a large number of apparently unpromising nodes on a list of candidates for possible future expansion. For examples, see Doran (1967) and Harris's *bandwidth search* (article IIC3c). Another reason for pruning is as a restriction on a search that is otherwise blind. For example, a breadth-first search could be modified to choose between expansion and pruning for each node it considers. This pruning to control the search is also very important for problems in which all solutions, rather than just a single solution, must be found; for finding all solutions implies an exhaustive exploration of all unpruned parts of the search space. An example of a search for all solutions is the DENDRAL program (see Applications.Dendral).

Ordered State-space Search

An *ordered* or *best-first search*, as mentioned above, is one that always selects the most promising node as the next node to expand. The choice is ordinarily assumed to be global, that is, to operate on the set of all nodes generated but not yet expanded. A local choice would also be possible, however; for example, an *ordered depth-first search* would be one that always expands the most promising successor of the node last expanded.

The promise of a node can be defined in various ways. One way, in a state-space problem, is to estimate its distance from a goal node; another is to assume that the solution path includes the node being evaluated and estimate the length or difficulty of the entire path. Along a different dimension, the evaluation may consider only certain predetermined features of the node in question, or it may determine the relevant features by comparing the given node with the goal. In all these cases, the measure by which the promise of a node is estimated is called an *evaluation function*.

A basic algorithm for ordered state-space search is given by Nilsson (1971). The evaluation function is f^*; it is defined so that the more promising a node is, the smaller is the value of f^*. The node selected for expansion is one at which f^* is minimum. The state space is assumed to be a general graph.

The algorithm is as follows:

```
(1) Put the start node s on a list, called OPEN, of unexpanded nodes. Calculate
    f*(s) and associate its value with node s.
(2) If OPEN is empty, exit with failure; no solution exists.
(3) Select from OPEN a node i at which f* is minimum. If several nodes qualify,
    choose a goal node if there is one, and otherwise choose among them
    arbitrarily.
(4) Remove node i from OPEN and place it on a list, called CLOSED, of expanded
    nodes.
(5) If i is a goal node, exit with success; a solution has been found.
(6) Expand node i, creating nodes for all its successors. For every successor
    node j of i:
    (a) Calculate f*(j).
    (b) If j is neither in list OPEN nor in CLOSED, then add it to OPEN, with its
        f* value. Attach a pointer from j back to its predecessor i (in order
        to trace back a solution path once a goal node is found).
    (c) If j was already on either OPEN or CLOSED, compare the f* value just
        calculated for j with the value previously associated with the node.
        If the new value is lower, then (i) substitute it for the old value,
        (ii) point j back to i instead of to its previously found predecessor,
        and (iii) if node j was on the CLOSED list, move it back to OPEN.
(7) Go to 2.
```

Step 6c is necessary for general graphs, in which a node can have more than one predecessor. The predecessor yielding the smaller value of $f^*(j)$ is chosen. For trees, in which a node has at most one predecessor, step 6c can be ignored. Note that even if the search space is a general graph, the subgraph that is made explicit is always a tree since node j never records more than one predecessor at a time.

Breadth-first, uniform-cost, and depth-first search (Article C1, Blind State-space Search) are all special cases of the ordered search technique. For breadth-first search, we choose $f^*(i)$ to be the depth of node i. For uniform-cost search, $f^*(i)$ is the cost of the path from the start node to node i. A depth-first search (without a depth bound) can be obtained by taking $f^*(i)$ to be the negative of the depth of the node.

The purpose of ordered search, of course, is to reduce the number of nodes expanded as compared to blind-search algorithms. Its effectiveness in doing this depends directly on the choice of f^*, which should discriminate sharply between promising and unpromising nodes. If the discrimination is inaccurate, however, the ordered search may miss an optimal solution or all solutions. If no exact measure of promise is available, therefore, the choice of f^* involves a trade-off between time and space on the one hand and the guarantee of an optimal solution, or any solution, on the other.

Problem Types and the Choice of f^*

The measure of a node's promise--and consequently, the appropriateness of a particular evaluation function--depends on the problem at hand. Several cases can be distinguished by the type of solution they require. In one, it is assumed that the state space contains multiple solution paths with different costs; the problem is to find the optimal (i.e., minimum cost) solution. This first case is well understood; see Article C3b on the A^* algorithm.

The second situation is similar to the first but with an added condition: The problem is hard enough that, if it is treated as an instance of case one, the search will probably exceed bounds of time and space before finding a solution. The key questions for case two are (a) how to find good (but not optimal) solutions with reasonable amounts of search effort and (b) how to bound both the search effort and the extent to which the solution produced is less than optimal.

A third kind of problem is one in which there is no concern for the optimality of the solution; perhaps only one solution exists, or any solution is as good as any other. The question here is how to minimize the search effort--instead of, as in case two, trying to minimize some combination of search effort and solution cost.

An example of case three comes from theorem proving, where one may well be satisfied with the most easily found proof, however inelegant. A clear example of case two is the traveling-salesman problem, in which finding some circuit through a set of cities is trivial, and the difficulty, which is very great, is entirely in finding a shortest or close-to-shortest path. Most treatments, however, do not clearly distinguish between the two cases. A popular test problem, the 8-puzzle, can be treated as being in either class. For further discussion of cases two and three, see Article C3c, Relaxing the Optimality Requirement.

References

See Doran (1967), Feigenbaum (1969), Gaschnig (1977), Michie (1967), Michie & Ross (1970), Newell & Ernst (1965), Newell & Simon (1972), Nilsson (1971), Nilsson (1974), Pohl (1977), Sandewall (1971), and Simon & Kadane (1975).

C3b. A*--Optimal Search for an Optimal Solution

The A* algorithm, described by Hart, Nilsson, and Raphael (1968), addresses the problem of finding a minimal cost path joining the start node and a goal node in a state-space graph. This problem subsumes the problem of finding the path between such nodes containing the smallest number of arcs. In the latter problem, each arc (representing the application of an operator) has cost 1; in the minimal cost path problem, the costs associated with arcs can be arbitrary. Historically, the predecessors of A* include Dijkstra's algorithm (1959) and Moore's algorithm (1959). A class of algorithms similar to A* is used in operations research under the name *branch-and-bound* algorithms (see Hall, 1971; Hillier & Lieberman, 1974; Lawler & Wood, 1966; and Reingold, Nievergelt, & Deo, 1977).

The algorithm used by A* is an *ordered state-space search* (Article C3a). Its distinctive feature is its definition of the *evaluation function* f*. As in the usual ordered search, the node chosen for expansion is always one at which f* is minimum.

Since f* evaluates nodes in light of the need to find a minimal cost solution, it considers the value of each node n as having two components: the cost of reaching n from the start node, and the cost of reaching a goal from node n. Accordingly, f* is defined by

$$f^*(n) = g^*(n) + h^*(n)$$

where g* estimates the minimum cost of a path from the start node to node n, and h* estimates the minimum cost from node n to a goal. The value f*(n) thus estimates the minimal cost of a solution path passing through node n. The actual costs, which f*, g*, and h* only estimate, are denoted by f, g, and h, respectively. It is assumed that all arc costs are positive.

The function g*, applied to a node n being considered for expansion, is calculated as the actual cost from the start node s to n along the cheapest path found so far by the algorithm. If the state space is a tree, then g* gives a perfect estimate since only one path from s to n exists. In a general state-space graph, g* can err only in the direction of overestimating the minimal cost; its value is adjusted downward if a shorter path to n is found. Even in a general graph, there are certain conditions (mentioned below) under which g*(n) can be shown to be a perfect estimate by the time node n is chosen for expansion.

The function h* is the carrier of *heuristic information* and can be defined in any way appropriate to the problem domain. For the interesting properties of the A* algorithm to hold, however, h* should be nonnegative, and it should never overestimate the cost of reaching a goal node from the node being evaluated. That is, for any such node n it should always hold that h*(n) is less than or equal to h(n), the actual cost of an optimal path from n to a goal node. This last condition is called the *admissibility condition*.

Admissibility and Optimality of A*

It can be shown that if h* satisfies the admissibility condition and if, in addition, all arc costs are positive and can be bounded from below by a positive number, then A* is guaranteed to find a solution path of minimal cost if any solution path exists. This property is called the property of *admissibility*.

Although the admissibility condition requires h^* to be a lower bound on h, it is to be expected that the more nearly h^* approximates h, the better the algorithm will perform. If h^* were identically equal to h, an optimal solution path would be found without ever expanding a node off the path (assuming only one optimal solution exists). If h^* is identically zero, A^* reduces to the blind uniform-cost algorithm (Article C1). Two otherwise similar algorithms, say A1 and A2, can be compared with respect to their choices of the h^* function, say $h1^*$ and $h2^*$. Algorithm A1 is said to be *more informed than* A2 if, whenever a node n (other than a goal node) is evaluated,

$$h1^*(n) > h2^*(n) .$$

On this basis an *optimality* result for A^* can be stated: If A and A^* are admissible algorithms such that A^* is more informed than A, then A^* never expands a node that is not also expanded by A. A proof (correcting the proof given in Nilsson, 1971) appears in Gelperin (1977).

Optimality and Heuristic Power

The sense in which A^* yields an optimal search has to do only with the number of nodes it expands in the course of finding a minimal-cost solution. But there are other relevant considerations. First, the difficulty of computing h^* also affects the total computational effort. Second, it may be less important to find a solution whose cost is absolutely minimum than to find a solution of reasonable cost within a search of moderate length. In such a case one might prefer an h^* that evaluates nodes more accurately in most cases but sometimes overestimates the distance to a goal, thus yielding an inadmissible algorithm. (See Article C3c.) The choice of h^* and the resulting *heuristic power* of the algorithm depend upon a compromise among these considerations.

A final question one might consider is the number of node expansions, as opposed to the number of distinct nodes expanded by A^*. The two totals will be the same provided that whenever a node n is expanded (moved to the CLOSED list), an optimal path to n has already been found. This condition is always satisfied in a state-space tree, where $g^*(n) = g(n)$ necessarily. It will also be satisfied in a general state-space graph if a condition called the *consistency assumption* holds (see Hart, Nilsson, & Raphael, 1968). The general idea of the assumption is that a form of the triangle inequality holds throughout the search space. Specifically, the assumption is that for any nodes m and n, the estimated distance $h^*(m)$ from m to a goal should always be less than or equal to the actual distance from m to n plus the estimated remaining distance, $h^*(n)$, from n to a goal. For an h^* not satisfying the consistency assumption on a general state-space graph, Martelli (1977) has shown that A^* is not optimal with respect to the number of expansions and has given an algorithm that runs more efficiently under these circumstances.

References

See Dijkstra (1959), Gelperin (1977), Hall (1971), Hart, Nilsson, & Raphael (1968), Hart, Nilsson, & Raphael (1972), Hillier & Lieberman (1974), Lawler & Wood (1966), Martelli (1977), Moore (1959), and Reingold, Nievergelt, & Deo (1977).

C3c. Relaxing the Optimality Requirement

The A* algorithm (C3b) is an ordered state-space search using the evaluation function $f^* = g^* + h^*$. If the appropriate conditions are met, including most importantly the *admissibility condition*, that the estimate $h^*(n)$ is always less than or equal to $h(n)$, then A* is guaranteed to find an optimal solution path if one exists. Again under suitable conditions, the performance of A* is optimal in comparison with other similarly defined admissible algorithms. Still, several questions remain:

(1) One may be more concerned with minimizing search effort than with minimizing solution cost. Is $f^* = g^* + h^*$ an appropriate evaluation function in this case?

(2) Even if solution cost is important, the combinatorics of the problem may be such that an admissible A* cannot run to termination. Can speed be gained at the cost of a bounded decrease in solution quality?

(3) It may be hard to find a good heuristic function h^* that satisfies the admissibility condition; with a poor but admissible heuristic function, A* deteriorates into blind search. How is the search affected by an inadmissible heuristic function?

Minimizing Search Effort

An approach to the first question can be stated as follows. The reason for including g^* in the evaluation function is to add a breadth-first component to the search; without g^*, the evaluation function would estimate, at any node n, the remaining distance to a goal and would ignore the distance already covered in reaching n. If the object is to minimize search effort instead of solution cost, one might conclude that g^* should be omitted from the evaluation function. An early heuristic search algorithm that did just this was Doran and Michie's Graph Traverser (Doran & Michie, 1966; Doran, 1967); the evaluation function used was of the form $f^* = h^*$, and the object was to minimize total search effort in finding solutions to the 8-puzzle and other problems. A generalization covering the Graph Traverser algorithm, A*, and others has been defined by Pohl (1969, 1970a, 1970b) as the Heuristic Path Algorithm (HPA). This algorithm gives an ordered state-space search with an evaluation function of the form

$$f^* = (1 - w)g^* + wh^*$$

where w is a constant in [0, 1] giving the relative importance to be attached to g and h. Choosing w = 1 gives the Graph Traverser algorithm; w = 0 gives breadth-first search; and w = .5 is equivalent to the A* function $f^* = g^* + h^*$.

Pohl's results concerning HPA indicate that, at least in special cases, omitting g^* from the evaluation function is a mistake. One case is that in which h^* is the most accurate heuristic function possible: if $h^*(n) = h(n)$ at every node n, the evaluation function $f^* = h^*$ still expands no fewer nodes than $f^* = g^* + h^*$. The other case assumes a simplified state space, whose graph is an infinite m-ary tree, and assumes that the error in h^*--which may underestimate or overestimate h--is bounded by a nonnegative integer e. In this situation it is shown that the maximum number of nodes expanded with $f^* = h^*$ is greater than the maximum number expanded with $f^* = g^* + h^*$, and that the difference between the maxima is exponential in the error bound e. This analysis by Pohl is one of the earliest applications of oracle or adversary analysis for discovering worst-case algorithmic efficiency. As such it is an important precursor to work on NP-complete problems and their attempted solution by heuristics. (For a general introduction to NP-completeness see Aho, Hopcroft, & Ullman, 1974.)

The two functions $f^* = h^*$ and $f^* = g^* + h^*$ have not been analyzed with respect to their average-case, as opposed to worst-case, behavior. Pohl's empirical results suggest that ordered search may typically expand the fewest nodes, provided the h^* function is fairly good, if g^* is included but given less weight than h^*--that is, with w greater than .5 but less than 1. These results were obtained for the 15-puzzle, a task exactly like the 8-puzzle except that it uses 15 tiles in a 4 x 4 array.

For problems that differ from the 15-puzzle, in that some states lead to dead ends rather than only to longer solutions, a somewhat different approach has been taken recently by Simon and Kadane (1975). Whereas the evaluation functions $f^* = g^* + h^*$ and $f^* = h^*$ are based on the estimated solution cost at a given node, Simon and Kadane propose that the function should also take explicit account of the probability that the node is in fact on a solution path. With such a function, an expected long search with high probability of success could readily rate just as favorably as one that is potentially shorter but which has a higher chance of failing.

Solution Quality and Heuristic Error

The second question, of speed vs. solution quality, has been studied by Pohl (1973, 1977) and Harris (1973, 1974). Harris's work concerns the third question (inadmissible heuristic functions) as well, as do Pohl's results summarized above. Both Harris and Pohl consider the *traveling-salesman problem*, which is NP-complete (Karp, 1972).

Pohl's approach is a further generalization of the HPA evaluation function: Now $f^*(n) = g^*(n) + w(n)h^*(n)$. That is, the relative weight w to be attached to g^* and h^* is no longer constant; the function w(n), which may be greater than or equal to 1, is defined to vary with the depth of node n. This approach is called *dynamic weighting*. With a definition of w that weights h^* less heavily as the search goes deeper, and with the assumption that h^* is a lower bound on h, Pohl shows that HPA will find a solution to the traveling-salesman problem whose cost is bounded by the ratio

$$\frac{\text{cost of tour found}}{\text{cost of optimal solution}} < 1 + e$$

where e is a constant in [0,1) which appears in the definition of w.

Dynamic weighting was tested on an instance of the traveling-salesman problem, known as the Croes problem, which involves 20 cities and has a known optimal solution cost of 246. An admissible A*--which produces an optimal solution if it produces any--had still not terminated after expanding 500 nodes. With dynamic weighting, however, together with an appropriate choice of e and the same h^* function, a solution with cost 260 was found by expanding only 53 nodes.

Harris's approach, called *bandwidth search*, is somewhat different from Pohl's. It assumes that no good h^* function satisfying the admissibility condition is available. In its place, he introduces the *bandwidth condition*, which requires that for all non-goal nodes n,

$$(1)\ h^*(n) \le h(n) + e$$

and

$$(2)\ h(n) - d \le h^*(n)\ .$$

It is assumed that h^* satisfies the *consistency assumption* (see Article C3b).

With respect to the first part of the condition, it can be shown that if h^* never overestimates the distance to a goal by more than e, the cost of a solution found by A^* will not exceed the cost of an optimal solution by more than e. With such an h^*, the algorithm is said to be *e-admissible*; and the goal it finds, *e-optimal*.

Once the bandwidth search finds some solution, a further application of condition (1) may show that the cost of the solution found is in fact closer than e to an optimal solution. This is possible because (a) the cost of the solution found is known, and (b) a lower bound on the cost of every other solution is the minimum, over all nodes n remaining on the OPEN list, of $f^*(n) - e$. If the difference between these two quantities is too big, the search can be continued until it finds a solution that is acceptably close to the optimum.

The second part of the bandwidth condition, condition (2), can be used to save storage space by dropping nodes from the OPEN list, without any risk of dropping a node that is in fact on an optimal path to a goal. Let node q be a node that, having a minimum value of f^*, has been selected for expansion. Then any node m may safely be dropped from OPEN if $f^*(m)$ is hopelessly big compared to $f^*(q)$. Specifically, it can be shown that all nodes m can be dropped if there is a node q such that

$$f^*(m) - (e + d) > f^*(q)\ .$$

Harris notes that it may be difficult to find a heuristic function h^* that satisfies both parts of the bandwidth condition. One may instead define two heuristic functions, one to order the search and one to determine which nodes can be dropped. Such functions, say $h1^*$ and $h2^*$, should then satisfy

$$(1')\ h1^*(n) \le h(n) + e$$

and

$$(2')\ h(n) - d \le h2^*(n)\ .$$

Using two such heuristic functions, Harris tested the bandwidth search on several instances of the traveling-salesman problem including the 20-city Croes problem mentioned above. Harris's results, including a comparison with A^* using an admissible heuristic function, are summarized below. The OPEN list was limited to 500 nodes.

	BANDWIDTH SEARCH		ADMISSIBLE SEARCH	
No. of cities	Quality of solution	Nodes expanded	Quality of solution	Nodes expanded
6	5-optimal	6		
6	optimal	14	optimal	18
11	optimal	14	none	500 open nodes
20	4-optimal	42	none	500 open nodes

Figure 1. Comparison of bandwidth search and admissible search.

References

See Aho, Hopcroft, & Ullman (1974), Doran & Michie (1966), Doran (1967), Harris (1973), Harris (1974), Karp (1972), Nilsson (1971), Pohl (1969), Pohl (1970a), Pohl (1970b), Pohl (1973), Pohl (1977), and Simon & Kadane (1975).

C3d. Bidirectional Search

Earlier articles in this chapter describe (a) heuristic state-space search methods using forward reasoning and (b) a blind state-space search combining forward and backward reasoning into a *bidirectional* algorithm. The kinds of problems to which a bidirectional state-space method applies are considered in Article C1; in general, it must be possible in these problems to search either forward, from the initial state toward the goal, or backward, from the goal toward the initial state. A bidirectional search pursues both lines of reasoning in parallel, growing two search trees and terminating when they meet. The motivation is that, in many cases, the number of nodes in a search tree grows exponentially with its depth; if a solution can be found by using two trees of half the depth, the search effort should be reduced significantly. Blind bidirectional search was in fact found to expand far fewer nodes than its unidirectional counterpart. A natural next question is whether heuristic bidirectional search can give still greater improvements in efficiency.

This question was investigated by Pohl (1969, 1971). Whereas his blind bidirectional algorithm used forward and backward uniform-cost search, his heuristic algorithm used forward and backward *ordered search*. Otherwise, the two algorithms differed mainly in their termination conditions. In both cases the termination condition was complicated by the fact that the algorithms were designed to find an *optimal* path between the start and goal nodes; they could be simplified if any path would do.

As *evalution functions*, Pohl's heuristic bidirectional algorithm used functions parallel to those of A*. For a node x in the forward search tree:

gs(x) measured the shortest path found so far from the start node, s, to x;

hs(x) estimated the minimum remaining distance from x to the terminal node, t; and

fs(x) = gs(x) + hs(x) was the evaluation function.

Similarly, for a node x generated in the backward search:

gt(x) measured the shortest path found so far from x to t;

ht(x) estimated the minimum distance from s to x; and

ft(x) = gt(x) + ht(x) was the evaluation function.

Constraints were placed on the heuristic functions hs and ht, corresponding to the admissibility condition and the consistency assumption of A*, in order to guarantee the optimality of the solution.

Pohl's results, in experiments using bidirectional heuristic search on the 15-puzzle, were disappointing. It was hoped that the search trees rooted at the start and goal nodes would meet near the middle of the solution path. In blind search, this had happened necessarily because both trees were expanded breadth-first. (Recall that uniform-cost search is a generalization of the breadth-first algorithm.) In the heuristic case, however, th search in each direction was narrowed. Since each problem had many alternate solutions, the typical outcome was that both search trees grew to include nearly complete, but different, solution paths before intersecting.

Several ideas have been advanced for forcing the trees to meet earlier while retaining the benefit of heuristic information (Pohl, 1971; Kowalski, 1972; de Champeaux & Sint, 1975, 1977; Pohl, 1977). One that has been tested is that of Champeaux and Sint, which redefines the heuristic functions hs and ht as follows:

> Let T-OPEN be the list of unexpanded nodes of the backward search tree. For a node x in the forward search tree, hs(x) estimates the minimum distance from x to the goal t by way of some node y in T-OPEN. That is, hs(x) is the minimum, over all nodes y on T-OPEN, of the estimated distance from x to y plus gt(y), the length of the shortest known path from y to the goal.

The function ht is defined analogously. The authors reported, for the same problems Pohl had used, that the algorithm generally produced shorter solution paths, with fewer nodes expanded, and that the search graphs now did meet near the middle of the search space. Unfortunately, however, hs and ht were so expensive to compute--since for each node x to be expanded, its distance must be estimated to every node y on the opposite OPEN list--that the algorithm still ran much more slowly than unidirectional heuristic search.

References

See de Champeaux & Sint (1975), de Champeaux & Sint (1977), Kowalski (1972), Pohl (1969), Pohl (1971), and Pohl (1977).

C4. Heuristic Search of an AND/OR Graph

This article returns to the problem of searching an AND/OR graph, as opposed to an ordinary state-space graph. The distinction between the two is the presence of AND nodes, which add conceptual complications to the search problem. Each node of the AND/OR graph represents a goal to be achieved. It will be assumed throughout that reasoning is backward, from an initial goal (the root) toward an equivalent set of subgoals, all of which have immediate solutions. On this assumption, an AND/OR graph constitutes (in the terminology of this chapter) a *problem-reduction representation*. This identification gives another way of stating the distinction between problem-reduction and state-space representations: State-space operators always take exactly one input and produce exactly one output; a problem-reduction operator also takes a single input but may produce multiple outputs (see Section B).

To put the matter further into perspective, one may also conceive of searching an AND/OR graph in the forward direction--from the primitive problems, whose solutions are already known, toward the problem one actually wishes to solve. Just such a graph search is that typically conducted by a resolution theorem-prover, as it brings together two or more axioms or previous conclusions and applies to them an operator yielding one new deduction as its result. (See Theorem Proving.) *Forward reasoning* in an AND/OR graph, then, would be distinguished from a state-space search by the presence of multiple-input, single-output operators. For further discussion, including an algorithm for *bidirectional search* of an AND/OR graph, see Kowalski (1972); see also Martelli and Montanari (1973).

The search of an AND/OR graph using backward reasoning raises numerous problems. Previous articles (B2 and C2) have considered

(a) what constitutes a solution subgraph of an AND/OR graph, and

(b) blind search algorithms for finding a solution subgraph.

This article considers three additional problems:

(c) What might one mean by an optimal solution subgraph?

(d) How can *heuristic information* be brought to bear on the search for an optimal solution?

(e) What limitations are there on AND/OR graphs and the associated search algorithms as general tools for problem solving?

The Definition of an Optimal Solution

A solution of an AND/OR graph is a subgraph demonstrating that the start node is solved. As in a state-space search, one may ask for a solution of minimal cost. The cost of a solution tree can be defined in either of two ways (Nilsson, 1971):

The *sum cost* of a solution tree is the sum of all arc costs in the tree.

The *max cost* of a solution tree is the sum of arc costs along the most expensive path from the root to a terminal node.

For example, if every arc in the solution tree has cost 1, then the sum cost is the number of arcs in the tree, and the max cost is the depth of the deepest node.

If the entire search space had been explored, then an optimal solution tree could be constructed and its cost measured as follows. Let c(n,m) be the cost of the arc from node n to a successor node m. Define a function h(n) by:

If n is a terminal node (a primitive problem), then h(n) = 0.

If n has OR successors, then h(n) is the minimum, over all its successors m, of c(n,m) + h(m).

If n has AND successors and sum costs are used, then h(n) is the summation, over all successors m, of c(n,m) + h(m).

If n has AND successors and max costs are used, then h(n) is the maximum, over all successors m, of c(n,m) + h(m).

If n is a nonterminal node with no successors, then h(n) is infinite.

According to this definition, h(n) is finite if and only if the problem represented by node n is solvable. For each solvable node n, h(n) gives the cost of an optimal solution tree for the problem represented by node n. If s is the start node, then h(s) is the cost of an optimal solution to the initial problem.

Consider, for example, the AND/OR tree of Figure 1, with arc costs as indicated. Each node without successors is marked t or u according to whether it is terminal or unsolvable.

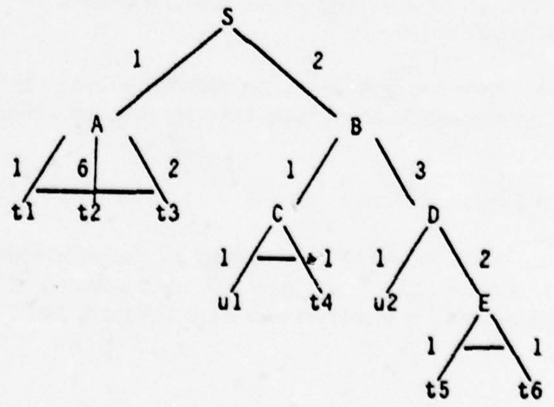


Figure 1. An AND/OR tree.

If *sum costs* are used, the values of h are as shown in Figure 2, and the optimal solution is the subgraph comprising nodes S, B, D, E, t5, and t6. The abbreviation *inf* denotes infinity.

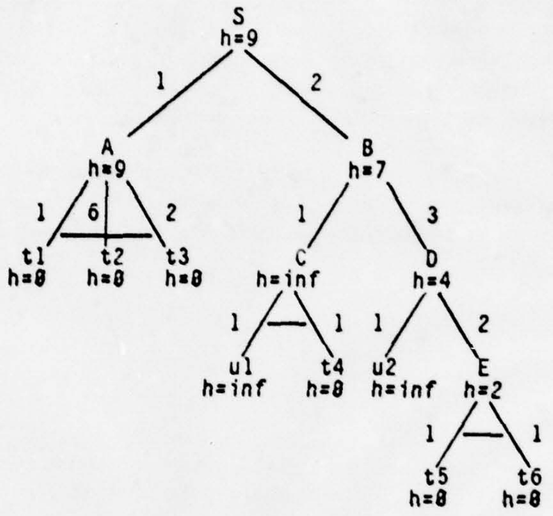


Figure 2. Sum costs.

If *max costs* are used, then the values of h are as shown in Figure 3, and the optimal solution is the subgraph comprising nodes S, A, t1, t2, and t3.

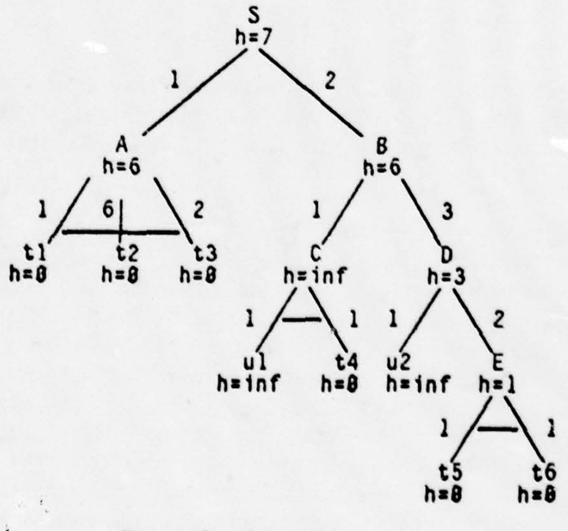


Figure 3. Max costs.

Ordered Search Algorithms for an AND/OR Graph

In an ordered state-space search, one may use an *evaluation function* f* that, applied to node n, returns the estimated minimum cost of a solution path passing through node n. The next node expanded is always one at which f* is minimum--that is, one extends the most promising potential solution path. The successors of node n are new nodes, but one could just as well think of them as new potential solution paths, each differing from a parent (potential solution path) by the inclusion of one more step.

In the extension of heuristic search to AND/OR graphs, there is no longer a one-to-one correspondence between the choice of a node to expand and the choice of a potential solution to be extended. Consider, for example, the search graph of Figure 4.

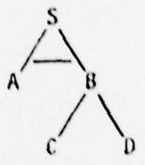
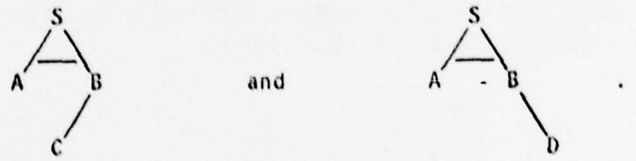


Figure 4. An AND/OR graph containing two potential solution trees.

Since C and D are OR nodes, an actual solution of node S will contain only one of them. To expand node A is thus to extend two *potential solution trees*,



Conversely, a decision to extend the potential solution tree on the left can be carried out by expanding either node A or node C. One must be clear, therefore, about what kind of object the expansion process is to apply to. This decision will affect the definition of the evaluation function.

Nilsson's algorithm. An approach taken by Nilsson (1969, 1971) selects individual nodes to expand by a two-step process: First, identify the most promising potential solution tree; then choose a node within that tree for expansion. To accomplish the first step, an evaluation function h* is defined at every node *n* of the tree that has not been shown to be unsolvable. This function is an estimate of h(*n*); that is, it estimates the cost of an optimal solution to the problem at node n. If n is known to be a terminal node, then by definition h*(n) = h(n) = 0. Otherwise, if n has not yet been expanded, then the estimate must be based on whatever heuristic information is available from the problem domain. For example, in the search tree of Figure 4, h* would provide heuristic estimates of the cost of solving nodes A, C, and D. The following rule then permits h* to be computed for each node whose successors have already been generated (and to be recomputed as the search tree is expanded):

> If *n* has OR successors m, then h*(n) is the minimum, over these successors, of c(n,m) + h*(m).

If n has AND successors m and sum costs are used, then $h^*(n)$ is the summation, over these successors, of $c(n,m) + h^*(m)$.

If n has AND successors m and max costs are used, then $h^*(n)$ is the maximum, over these successors, of $c(n,m) + h^*(m)$.

Finally, the most promising potential solution tree, T, is defined in terms of h^*:

The start node s is in T.

If the search tree (the part of the search space generated so far) contains a node n and AND successors of n, then all these successors are in T.

If the search tree contains a node n and OR successors m of n, then one successor m is in T such that $c(n,m) + h^*(m)$ is minimal.

The estimated cost of T is $h^*(s)$. If all the other potential solution trees for the same search tree were constructed, it would be found that T is one for which $h^*(s)$ is minimal.

An ordered-search algorithm for an AND/OR tree can now be stated as follows:

```
(1) Put the start node, s, on a list, OPEN, of unexpanded nodes.
(2) From the search tree constructed so far (initially, just s), compute the most
    promising potential solution tree T.
(3) Select a node n that is on OPEN and in T. Remove node n from OPEN and
    place it on a list called CLOSED.
(4) If n is a terminal node, then
    (a) Label node n solved.
    (b) If the solution of n makes any of its ancestors solved, label these
        ancestors solved.
    (c) If the start node is solved, exit with T as the solution tree.
    (d) Remove from OPEN any nodes with a solved ancestor.
(5) Otherwise, if node n has no successors (i.e., if no operator can be applied),
    then
    (a) Label node n unsolvable.
    (b) If the unsolvability of n makes any of its ancestors unsolvable, label
        all such ancestors unsolvable as well.
    (c) If the start node is labeled unsolvable, exit with failure.
    (d) Remove from OPEN any nodes with an unsolvable ancestor.
(6) Otherwise, expand node n, generating all its immediate successors and, for
    each successor m representing a set of more than one subproblem,
    generating successors of m corresponding to the individual subproblems.
    Attach, to each newly generated node, a pointer back to its immediate
    predecessor, and compute $h^*$ for each newly generated node. Place all the
    new nodes that do not yet have descendants on OPEN. Finally, recompute
    $h^*(n)$ and $h^*$ at each ancestor of n.
(7) Go to (2).
```

The ordered-search algorithm can be shown to be *admissible*--that is, it will find a minimum-cost solution tree if any solution exists--provided that: (a) $h^*(n)$ is less than or equal to h(n) for each open node n, and (b) all arc costs are greater than some small positive number d. The efficiency of the algorithm, however, depends both on the accuracy of h* and on the implementation of step 3, in which, having found the most promising potential solution tree to expand, one must decide to expand a specific node within that tree. If the partial tree T is in fact part of an optimum solution, the choice is immaterial. If it is not, however, then the best node to expand would be the one that will earliest reveal the error.

Chang and Slagle's algorithm. A different approach has been taken by Chang and Slagle (1971). Here the objects expanded are potential solution graphs. A *tip node* in such a graph is any node that does not yet have successors. To expand the potential solution graph, one expands all its nonterminal tip nodes at once and then forms all the new potential solution graphs that result. Each graph is represented on the OPEN list by the conjunction of its tip nodes, representing a set of subproblems to which the start node can be reduced.

For example, suppose that expansion of the initial graph, consisting of only the start node S, shows that S can be reduced to problems A and B or to problem C. The OPEN list then becomes (A&B, C). Assume that A&B is selected for expansion, that A can be reduced to D or E, and that B can be reduced to F or G. There are four new potential solution trees, and the OPEN list is now (D&F, D&G, E&F, E&G, C). The search succeeds when it selects for expansion a potential solution graph represented by a conjunction of nodes all of which are terminal.

The Chang and Slagle approach assimilates AND/OR graph search to the problem of state-space search. Each distinct conjunction of problems to be solved corresponds to a distinct state of a state-space graph. The evaluation function used, f*, is also parallel to the function used in A*: It is defined by $f^* = g^* + h^*$, where g* measures the cheapest way found so far to reduce the start node to a given conjunction of subproblems and h* estimates the minimum remaining cost of a graph sufficient to solve all those subproblems.

The treatment of AND/OR graph search as an instance of state-space search has several consequences. One is that the search of a general AND/OR graph, as opposed to an AND/OR tree, now raises no special problems. Another is that the algorithm can be shown (Chang & Slagle, 1971), under appropriate conditions, to be not only admissible but also optimal with respect to the number of potential solution graphs expanded. It does not, however, appear to be optimal (in some reasonable sense of that term) in comparison with algorithms that expand only one node at a time (see Kowalski, 1972).

Interdependent Subproblems

The discussion so far has assumed that whenever the start node is reduced to a conjunction of subproblems, all subproblems can be solved independently, so that the solution to one has no effect on the solution to any other. This assumption is frequently unjustified, and much of the chapter on Planning explores ways of dealing with interacting subproblems. Two kinds of examples, given by Levi and Sirovich (1975, 1976) with explicit reference to the AND/OR graph formalism, are: (a) problems requiring consistent binding of variables and (b) problems involving the expenditure of scarce resources.

An illustration of the former is the well-known problem of showing that there exists a fallible Greek, given that the entire search space is as follows:

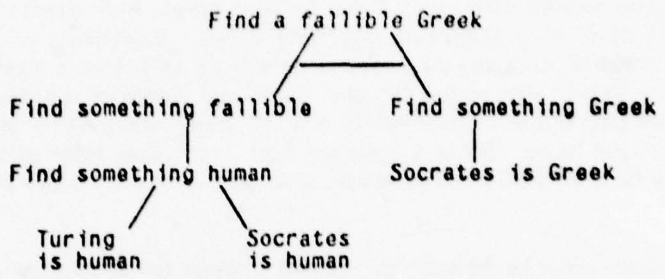


Figure 5. An AND/OR graph requiring consistent binding of the variable "something."

An algorithm like Nilsson's fails here for two reasons. First, it has no mechanism for discovering that "Turing is human" and "Socrates is Greek" fail to constitute a solution. Second, even if such a mechanism were introduced, the algorithm has no means for undoing the solution to a subproblem once it has been solved. If "Turing is human" is the first problem found to be primitive, then "Find something human" and "Find something fallible" are marked solved; "Socrates is human" is removed from the OPEN list as no longer in need of consideration; and "Find something Greek," using the previous value of "something," then becomes unsolvable.

An example of the second type of problem is the following: Show that John can seduce the actress, given that seducing the actress can be reduced to getting a car and getting a yacht; and that John has $5000, a car costs $5000, and a yacht costs $5000. Here either of the algorithms given above would wrongly conclude that John can seduce the actress. A variant of the scarce resource problem arises in robot planning tasks (such as those performed by STRIPS, Article D5), where application of an operator representing a robot action solving one subproblem may make inapplicable the operator needed to solve another subproblem.

To handle problems of these kinds, Levi and Sirovich define a *generalized AND/OR graph*, which differs most importantly from an ordinary AND/OR graph in that reduction operators are permitted to take two or more nodes as input. For example, let R be a resource that can be used only once. Then if, in the standard formulation, the original problem is to accomplish P1 and P2, the problem is reformulated as P1 & P2 & R. Suppose the following reduction operators are available (where -> means "can be reduced to" and T denotes a trivial problem):

1) S -> P1 & P2 & R
2) P1 & R -> T
3) P1 -> P3
4) P2 & R -> P3
5) P3 -> T
6) R -> T

Then there is only one solution, which is achieved using operators 1, 3, 4, and 5.

In the ordered search of a generalized AND/OR graph, the objects placed on the OPEN list are potential solution graphs, not individual nodes. Expansion of a potential solution graph (PSG) consists of applying all possible operators to obtain a new set of PSGs, each differing from its parent by virtue of one additional operator application. If the same subproblem occurs more than once within a PSG, each occurrence is represented by a separate node. If the same PSG is generated more than once, later occurrences are simply discarded. Since distinct PSGs are retained, alternate solutions to the same subproblem are available.

As in the usual ordered search, the object chosen for expansion next is always one where the evaluation function is minimum. The evaluation function is h^*; for each PSG, it is computed similarly to the h^* of Nilsson's algorithm. The value of each potential solution graph is then the evaluation of the start node, $h^*(s)$, as computed for that graph. Both *admissibility* and *optimality*--the latter with respect to the number of PSGs expanded--can be shown.

References

See Chang & Slagle (1971), Kowalski (1972), Levi & Sirovich (1975), Levi & Sirovich (1976), Martelli & Montanari (1973), Nilsson (1969), and Nilsson (1971).

C5. Game Tree Search

C5a. Minimax Procedure

The Minimax Formalism

The minimax procedure is a technique for searching game trees (Article B3). As a first example, Figure 1 gives a simple game tree to which the procedure may be applied. Each node represents a position in the game. Nonterminal nodes are labeled with the name of the player, A or B, who is to move from that position. It is A's turn, and the problem is to find his best move from position 1. Exactly three moves remain in the game. Terminal nodes are marked with their value to player A by the words "win," "lose," or "draw."

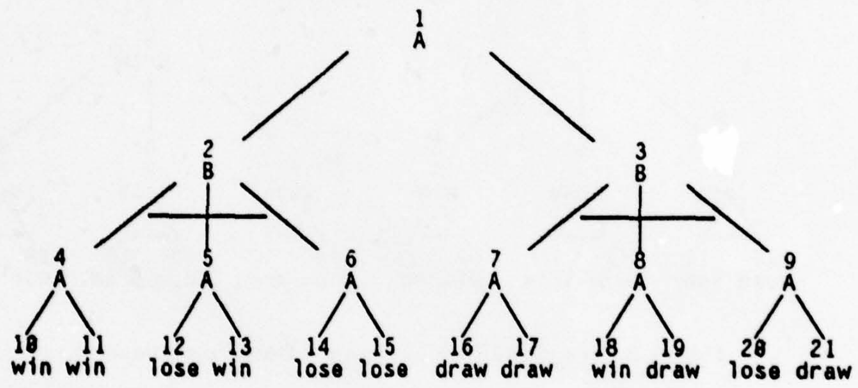


Figure 1. A game tree from the standpoint of player A, who is to move next.

According to the minimax technique, player A should move to whichever one of positions 2 or 3 has the greater value to him. Given the values of the terminal positions, the value of a nonterminal position is computed, by backing up from the terminals, as follows:

> The value to player A of a node with OR successors (a node from which A chooses the next move) is the maximum value of any of its successors.
>
> (1)
>
> The value to A of a node with AND successors (a node from which B chooses the next move) is the minimum value of any of its successors.

In the example, node 2 evaluates to a loss for A (since B can then force a loss by moving to node 6), and node 3 evaluates to a draw (since the best B can then do is move to node 7 or 9). It will be noted that the prediction of the opponent's behavior assumes he is also using minimax: in evaluating a node with AND successors, A must assume that B will make his best possible move. The technique ignores the possibility that B might overlook his chance for a

60

sure win if A goes to node 2. Similarly, it supplies no basis on which B might choose to move to node 9 in preference to node 7.

Because of the way in which nodes are evaluated, player A (whose viewpoint the tree represents) is often called MAX, and player B, MIN. The names PLUS and MINUS are also sometimes used. If the tree of Figure 1 were to be evaluated from MIN's standpoint instead of MAX's, it would appear as in Figure 2. The AND and OR nodes are reversed, and the value of each node to MIN is the opposite of its value to MAX.

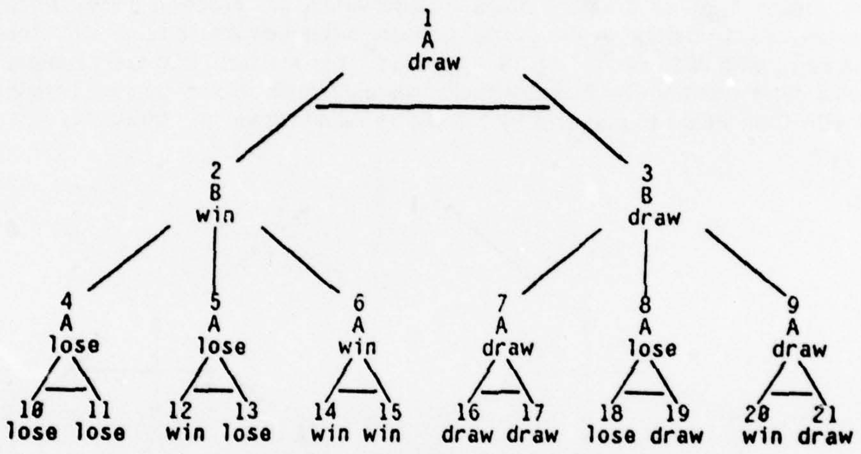


Figure 2. The game tree of Figure 1 from B's standpoint.

The Negmax Formalism

Knuth and Moore (1975) have given a game-tree representation that unifies Figures 1 and 2 and conveniently permits a single procedure to return optimal moves for both players A and B. In this representation, the value given each node is its value to the player whose turn it would be to move at that node. If n is a terminal node, its value is an integer denoted f(n). (The value of n to the other player is -f(n).) The value of every node is then returned by a function F defined as follows:

$F(n) = f(n)$, if n has no successors;
$F(n) = \max \{-F(n1), \ldots ,-F(nk)\}$, if n has successors $n1, \ldots ,nk$.

The best move for either player is then to a node with maximum value; that is, the player whose turn it is at node n should move from node n to a node ni with $- F(ni) = F(n)$. This formulation, which is equivalent to minimax, is called *negmax*. The tree it produces for the game of Figures 1 and 2 is shown in Figure 3. The numerical value of a win is assumed to be +1; of a loss, -1; and of a draw, 0.

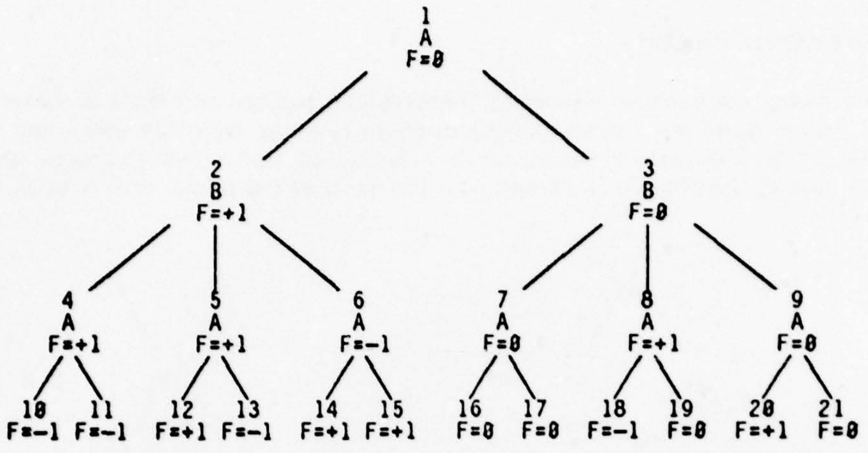


Figure 3. The game tree of Figure 1 in NEGMAX notation.

Searching a Partial Game Tree

In the above descriptions of the minimax and negmax algorithms, it was assumed that a complete game tree had already been generated. For most games, however, the tree of possibilities is far too large to be generated fully and searched backward from the terminal nodes for an optimal move. An alternative is to generate a reasonable portion of the tree, starting from the current position; make a move on the basis of this partial knowledge; let the opponent reply; and then repeat the process beginning from the new position. A "reasonable portion of the tree" might be taken to mean all legal moves within a fixed limit of depth, time, or storage, or it might be refined in various ways. For discussion of the refinements, see article C5c.

Once the partial tree exists, minimaxing requires a means for estimating the value of its *tip nodes*, that is, the nodes of the partial tree without successors. A function assigning such a value is called a *static evaluation function*; it serves a purpose comparable to that of the heuristic function h^* used in Nilsson's ordered search of an AND/OR tree (Article C4). If the partial game tree contains any nodes that are terminal for the entire tree, the static evaluation function conventionally returns positive infinity for a win, negative infinity for a loss, and zero for a draw. At other tip nodes, the function has a finite value which, in the minimax formulation, is positive for positions favorable to MAX and negative at positions favorable to MIN. The minimax procedure then assigns *backed-up values* to the ancestors of the tip nodes in accordance with the rules given in (1) above. It is assumed that the backed-up evaluations give a more accurate estimate of the true value of MAX's possible moves than would be obtained by applying the static evaluation function directly to those moves and not looking ahead to their consequences.

References

See Knuth & Moore (1975), Nilsson (1971), Slagle (1971), and Winston (1977).

C5b. Alpha-beta Pruning

The minimax procedure described in Article C5a decides on a best move from node n, in a full or partial game tree, by evaluating every node in the tree that descends from node n. Frequently, this exhaustive evaluation is a waste of time. Two examples are shown in Figures 1 and 2. Each node is marked with the name of the player who is to move from that position.

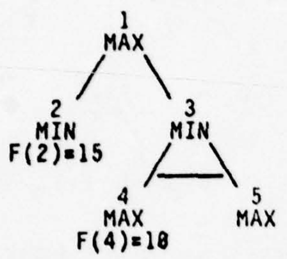


Figure 1. An alpha cutoff.

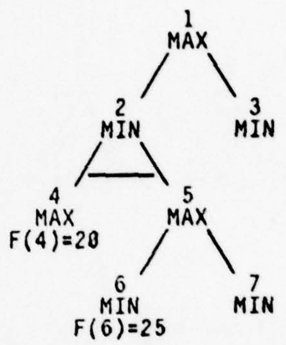


Figure 2. A beta cutoff.

In Figure 1, nodes 2 and 4 have been evaluated either by the static evaluation function or by backing up from descendants omitted from the figure. If MAX moves to node 2, he achieves a position whose estimated value is 15. If he moves to node 3, MIN can hold him to 10. Therefore, the value of node 3 is at most 10, so MAX should decide to move to node 2. The important point is that this decision can be made without evaluating node 5 or any of its possible descendants.

In Figure 2, node 4 has an estimated value to MAX of 20. When node 6 is evaluated at 25, it becomes clear that MIN should avoid moving to node 5. Node 2 can therefore be assigned a value of 20 without any need to evaluate node 7 or any of its descendants.

The *alpha-beta technique* for evaluating nodes of a game tree eliminates these unnecessary evaluations. If, as is usual, the generation of nodes is interleaved with their evaluation, then nodes such as the descendants of node 5 in Figure 1 and of node 7 in Figure 2 need never even be generated. The technique uses two parameters, alpha and beta. In Figure 1, the parameter alpha carries the lower bound of 15 on MAX's achievement from node 1; the elimination of node 5 is an *alpha cutoff*. In Figure 2, the parameter beta is set to 20 at node 4, representing an upper bound on the value to MAX of node 2; the elimination of node 7 is a *beta cutoff*. The procedure guarantees that the roct node of the tree will have the same final value as if exhaustive minimaxing were employed.

A concise statement of the alpha-beta procedure has been given by Knuth and Moore (1975). It uses their *negmax* representation in which both players are treated as wishing to maximize (see Article C5a). Figure 3 shows how Figures 1 and 2 are transformed in the negmax representation.

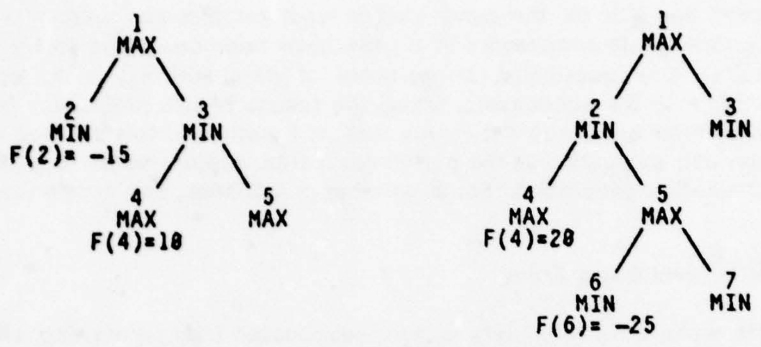


Figure 3. The NEGMAX representation of Figures 1 and 2.

To evaluate node 1 of either tree, the procedure is called with the parameters POSITION = node 1, ALPHA = negative infinity, and BETA = positive infinity. The static evaluation function is called f. The procedure, here called VALUE, is as follows:

```
INTEGER PROCEDURE value(POSITION p, INTEGER alpha, INTEGER beta)
    BEGIN
    INTEGER m, i, t, d
      determine the successor positions $p_1, p_2, \dots, p_d$
      of position p;
    IF d = 0 THEN value := f(p) ELSE
          BEGIN
          m := alpha;
          FOR i := 1 STEP 1 UNTIL d DO
                 BEGIN  t := -value ($p_i$, -beta, -m);
                 IF t > m THEN m := t;
                 IF m > beta or m = beta THEN GO TO done;
                 END;
          done: value := m;
          END;
    END;
```

For an intuitively developed LISP version of the alpha-beta procedure, see Winston (1977). An excellent review of the historical development of the technique appears in Knuth and Moore (1975).

Ordering of Successors

The degree to which the alpha-beta procedure represents an improvement in efficiency over straight minimaxing varies with the order in which successor nodes are evaluated. For example, no cutoff would occur in Figure 1 if node 3 were considered before node 2.

In general, it is desirable that the best successor of each node be the first one evaluated--that is, that the first move MAX considers be his best move, and that the first reply considered for MIN be the move that is best for MIN and worst for MAX. Several schemes for ordering the successors of a node have been described to try to achieve this state of affairs. One possibility, an example of *fixed ordering*, is to apply the static evaluation function to the successors, taking the results of this preliminary evaluation as an approximation of their expected backed-up values. A method of this sort will result in depth-first generation and evaluation of the partial game tree, subject to the depth bound or other criteria for terminating generation. For some other possibilities, see Article C5c.

Efficiency in Uniform Game Trees

Since the alpha-beta procedure is more complicated than minimaxing, although it yields the same result, one may inquire how great an increase it produces in search efficiency. Most theoretical results on this question deal with *uniform* game trees: A tree is said to be uniform if every tip node has depth d and every nontip node has exactly b successors. Here b is called the *branching factor* or *degree* of the tree.

The results reviewed below come from Knuth and Moore (1975) and, for the best case, Slagle and Dixon (1969). For other related work, see Fuller et al. (1973), Newborn (1977), and Baudet (1978).

The best case. *A uniform game tree of depth d and degree b contains exactly* b^d *tip* nodes, all of which must be examined by minimax. In the worst case, alpha-beta also must examine every tip node. In the best case, alpha-beta examines only about twice the square root of the number of tip nodes. More precisely, assuming the value of the root is not infinite, the number of tip nodes examined in the best case is

$$b^{[(d+1)/2]} + b^{[d/2]} - 1$$

(where square brackets represent the greatest integer function); and the nodes examined in the tree as a whole are precisely the *critical nodes*, defined as follows:

Type 1 critical nodes are the root node and all first successors of type 1 nodes.

Type 2 critical nodes are all further successors (except the first) of type 1 nodes and all successors of type 3 nodes.

Type 3 critical nodes are the first successors of type 2 nodes.

Figure 4 illustrates the distribution of critical nodes in a uniform tree of degree 3 and depth 3.

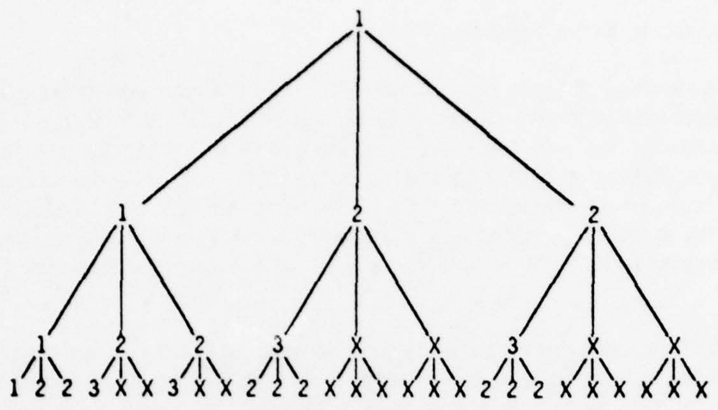


Figure 4. Distribution of critical nodes.

Knuth and Moore have shown that the best case occurs for a uniform tree if the best move is considered first at each critical node of types 1 and 2. Attempts to order the successors of type 3 positions contribute nothing to efficiency, since these successors are type 2 nodes, which must all be examined anyway.

Random uniform game trees. Knuth and Moore also show that the alpha-beta technique is optimal in the sense that no algorithm can evaluate any game tree by examining fewer nodes than alpha-beta does with an appropriate ordering of successors. Realistically, of course, one cannot expect to achieve the optimal successor ordering, since this would imply full knowledge of the game tree before it is generated. Assuming, therefore, that the tip nodes of the tree have distinct random values, Knuth and Moore show that the expected number of tip nodes examined, in evaluation of a uniform tree with branching factor b and depth d, has an asymptotic upper bound of

$$(b/(\log b))^d$$

as d goes to infinity.

Totally dependent uniform game trees. One other type of tree considered by Knuth and Moore, perhaps more realistic than the one in which tip nodes have random values, corresponds to games in which each move is critical: If a poor move is ever chosen, there is no way to recoup. The model is a uniform game tree that is *totally dependent*: For any two successors of node p, these successors can be labeled q and r so that every tip node descended from node q has greater value than any tip node descended from node r. In this type of tree, if the degree is at least 3, the expected number of tip positions examined is bounded by a constant (depending on the degree) multiplied by the number of tip nodes examined by the alpha-beta method in the best case.

References

See Baudet (1978), Fuller, Gaschnig & Gillogly (1973), Knuth & Moore (1975), Newborn (1977), Nilsson (1971), Slagle & Dixon (1969), Slagle (1971), and Winston (1977).

C5c. Heuristics in Game Tree Search

In the search of a game tree (Article B3), as in other kinds of search, there are various points at which heuristic information may be applied. The parallel is not exact, however. In one-person problem solving, the main uses for heuristic information are to decide which node to expand next, which operator to apply next, and, in some algorithms, which nodes to prune from the search tree. (See Article C3a.) In game-playing programs, these questions also exist, but with a shift in emphasis. In addition, some new questions arise: When should the search be terminated? How should a move be chosen on the basis of the search that has been made?

The simplest answers to these questions were described in Article C5a: Expand every node completely, in any convenient order and with no pruning, until every tip node represents a termination of the game. Then, working back from the end of the game, use the minimax procedure to find a winning line of play (if one exists), and follow this line of play throughout the game. Article C5b, Alpha-beta Pruning, described an improvement on this approach that yields the same final result with greater efficiency.

A program using only these basic techniques would play a theoretically perfect game; its task would be like searching an AND/OR tree for a solution to a one-person problem. For a simple game like tic-tac-toe (see Article B3), such a program would no doubt be feasible. For complex games, however, it has been recognized from the beginning that searching from the start of the game to its end would be impossible. In chess, for example, with around 30 legal moves from each position and about 40 moves for each player in a typical game, there are some $(30^2)^{40}$ or 10^{120} different plays of the game (Shannon, 1950).

Because of the magnitude of the search space in chess, checkers, and other nontrivial games, there is a major difference between programs that play such games and programs that use the methods of this chapter to solve nonadversary problems. The latter either find a solution or fail, having run out of time or space; much of the research assumes that some solution can be found and deals with how to guarantee that it is optimal or nearly optimal (see Section C3, Heuristic State-space Search). The question for a chess program, in contrast, is how to play a good game even though it has not found a solution to the problem of winning. Repeatedly the program must become committed to its next move long before the end of the game comes into view. Whether the move chosen is in fact part of a winning strategy is unknown until later in the game.

For a nontrivial game playing program, then, the issues listed at the beginning of this article are all aspects of a broader question: Can the basic search techniques, designed for seeking a guaranteed win, be successfully adapted to the problem of simply choosing the next move? In addition, one might well ask whether there are alternatives to search as the basis for move selection. Most of the work exploring these questions has been done in the specific domain of chess. In general, the discussion below is limited to chess programs and Samuel's checkers program (1963, 1967).

Alternatives to Search

An example of choosing a move on a basis other than search is the use of "book moves" in the opening of a chess game (see Frey, 1977, pp. 77-79). More generally, there is an emphasis in the recent computer chess literature on treating the problem of move choice as a problem of recognizing patterns on the board and associating appropriate playing methods with each pattern (e.g., Charness, 1977, p. 52; Bratko et al., 1978; Wilkins, 1979).

It is not expected, however, that search can be eliminated entirely from chess programs; even human players do some searching. Rather, the choice-of-move problem is seen as involving a tradeoff between the amount of specialized chess knowledge a program has and the amount of search it needs to do. (See, e.g., Berliner 1977c; Michie, 1977.) And there are limits on the amount of knowledge a program can be given: The combinatorics of chess preclude storing an exhaustive representation of the game; and even the knowledge possessed by chess masters, which greatly restricts search in human play, also remains very far from complete formalization.

The last section of this article reviews several programs that attempt to use human-like knowledge to eliminate most searching. The sections preceding it concern techniques used in programs in which search rather than knowledge is predominant.

Search-based Programs

The most successful game-playing programs so far have made search rather than knowledge their main ingredient. These include, among the earlier programs, Samuel's checkers program (1963, 1967), which came close to expert play; and Greenblatt's chess program (1967), which was the first to compete in tournaments and which earned a rating of 1400-1450, making it a Class C player. (Current classes of the United States Chess Federation are E through A, Expert, Master, and Senior Master. See Hearst, 1977, p. 171.) Notable later programs include the Soviet program KAISSA (Adelson-Velskiy et al., 1975), which won the first world computer chess championship in 1974, and Slate and Atkin's CHESS 4.5 (1977), whose current standing is mentioned below. (For general reviews of computer chess competition, see Berliner, 1978a; Mittman, 1977; and Newborn, 1975.)

All the programs referred to above follow the basic search paradigm formulated by Shannon in 1950. In its simplest form, which was called a Type A program, Shannon's paradigm made just two changes to the procedure mentioned above that calls for searching exhaustively all the way to the end of the game. First, the game tree was to be generated only to a fixed depth. Second, since the nodes at the depth limit would normally be nonterminal, a means of estimating the promise of these nodes was required. The estimate was to be given by a *static evaluation function*, whose values could then be backed up by minimaxing to determine the next move. After this move was made and the opponent had replied, the search process would be repeated beginning from the new position.

Shannon noted that a simple Type A program would play chess both badly and slowly. He suggested two directions for improvement in a Type A program, with which the program would become Type B. The general objectives were, first, to let the exploration of a line of play continue to a reasonable stopping point instead of invariably cutting it off at an arbitrary depth; and, second, to provide some selectivity about the lines of play considered, so that more time could be spent investigating strong moves and less on pointless ones.

Even a Type B program, Shannon concluded, seemed to rely too much on brute-force calculation rather than on knowledgeable analysis of the situation to choose a move.

Nevertheless, his proposals established a framework that most competitive game-playing programs have adopted. The framework raises a large number of interrelated issues, which are discussed in the following sections.

Static Evaluation

A *static evaluation function*, by definition, is one that estimates the value of a board position without looking at any of that position's successors. An ideal function would be one that reports whether the position leads to a win, a loss, or a draw (provided neither side makes a mistake). Even more informatively, the function might report the number of moves required to win, with an arbitrarily large value if no win is possible. But functions that can distinguish between winning and losing positions are known only for simple games; an example of such a function for the game Nim is given in Shannon (1950).

Where perfect evaluation functions are unavailable, the actual static evaluator must return an estimate. Unlike the *evaluation function* used in an ordinary state-space or AND/OR graph search (C3a, C4), the static evaluation function of a game-playing program does not normally attempt directly to estimate the distance to a win from the position evaluated. (For a proposal that the function should do just this, see Harris, 1974.) Instead, the function is usually a linear polynomial whose terms represent various features of the position, high values being given for features favorable to the program and low ones for those favoring the opponent. In chess, the most important feature is material, the relative value of each side's pieces on the board. Other typical features, familiar to chess players, include king safety, mobility, center control, and pawn structure.

The most extended treatment of evaluation functions in the literature is provided by Samuel (1963, 1967). For checkers, he concluded (1967, p. 611) that the optimal number of features to be used in the evaluation function was between twenty and thirty. Samuel's main interest was in machine learning; one approach he took was to provide his checkers program with a large set of features for possible use in the evaluation function and to let the program determine, as it gained playing experience, both which of these features should be included and what their relative weights should be. In a later version of the program, the emphasis was shifted to taking the interactions among features into account in evaluating positions. With this change, the evaluation function became nonlinear, and considerable improvement was reported in its quality as measured by the correlation with moves chosen in master play (Samuel, 1967; see also Griffith, 1974). For further discussion of Samuel's work, see Learning.

Reasonably accurate static evaluation, then, requires a rather complex function. But there is an important limit on the complexity that is feasible, especially for a program that plays in tournaments, under time limitations. As the total number of tip nodes in the search tree increases, the time available for evaluating any single tip node goes down. Thus Gillogly's chess program TECH (1972), which was intended as an experiment in how much could be accomplished on advanced machines by simple brute force search, and which generates up to 500,000 tip nodes even with alpha-beta pruning, uses material as the only factor in its static evaluations.

Backed-up Evaluation

The Shannon paradigm assumes that the step between static evaluation and the choice of a move is simply minimaxing: The program moves to any position with the best backed-up minimax value. This step is indeed very commonly used. But it is worth noting that, since the static evaluation function may be wrong, the minimax procedure no longer serves its original purpose of defining and identifying a move that is theoretically correct. Instead, minimaxing has itself become a heuristic for the choice of move. Several programs have therefore experimented with varying or supplementing the minimax procedure. Slagle and Dixon (1970), for example, in experiments with the game of kalah, compute the backed-up value of a node by taking into account not only the value of its best successor but also whether the node has several good successors or just one. Gillogly's TECH (1972), having computed minimax values on the basis of an extremely simple static evaluation, breaks ties between moves with equal minimax values by an analysis of features not considered by the evaluation function. Newell, Shaw, and Simon (1963a) set a value in advance that the search is expected to achieve; the first move found that meets this standard is made, and only if no move is good enough is the best minimax value used to determine the move (see also Newell & Simon, 1972).

Depth of Search

If perfect evaluation functions were available, a game-playing program could proceed at each turn by generating all legal moves, evaluating each of the resulting positions, and choosing the move leading to the best value. The reason for looking farther ahead is to compensate for errors in the static evaluation. The assumption is that, since static evaluation has a predictive aspect, there will be less room for mistaken prediction if a deep tree is generated before the evaluation function is applied.

The controlling fact about search depth is the combinatorial explosion. If the average number of legal moves from a position, the *branching factor*, is b, the game tree will have about b^d nodes at depth d. According to Shannon's estimate for chess, a complete tree carried to depth 6--3 moves for each player--would already have about one billion tip nodes. At the same time, Shannon noted, a world champion may occasionally look ahead, along a single line of play, to a depth as great as 15 or 20. More recently Hans Berliner, a former World Correspondence Chess Champion, has said he finds it necessary at least once in a game to look ahead to a depth of 14 or more (1974, p. I-8). The question, then, is how to get the needed depth, in the right places, without succumbing to the combinatorial explosion. An alternative question would be how to avoid the need for so deep a search. The remainder of this article concerns attempts to solve or at least alleviate these problems. First, however, experience with the use of depth bounds as such will be reviewed.

Fixed-depth search with extensions for quiescence. The simplest lookahead procedure, which was called for by Shannon's Type A strategy, is to set a fixed depth, or *ply*, to which the game tree is to be generated and to apply the static evaluation function only to nodes at this depth. Thus a 4-ply search would statically evaluate the positions reached after exactly two turns for each player. There are serious drawbacks in this procedure, as Shannon observed, and it was used only in very early programs (Kister et al., 1957; Bernstein et al., 1959). For example, a chess evaluation function based mainly on material cannot return a realistic value if at the depth limit the players happen to be halfway through an exchange of pieces. The concept of a *quiescent* or *dead* position was introduced to get around such difficulties (Shannon, 1950; see also Turing, 1953): Search would be extended beyond the normal limit, from nonquiescent positions only, until all tip nodes were relatively stable or perhaps until some absolute depth-bound had been reached.

This introduction of a quiescence search was one of the two features that changed a program, in Shannon's terminology, from Type A to Type B. On Shannon's suggested definition, a position was considered nonquiescent if "any piece is attacked by a piece of lower value, or by more pieces than defences or if any check exists on a square controlled by opponent" (1950, p. 271). Many programs have adopted a similar definition, with the result that the only moves examined beyond the normal limit are checks and immediate captures (e.g., Gillogly, 1972; Adelson-Velskiy et al., 1975; Slate & Atkin, 1977). If such a quiescence search is combined with considering all legal moves down to the normal depth limit, the program is still called Type A in current terminology (e.g., Berliner, 1978a).

The horizon effect. Searching to an arbitrarily limited depth, even with extensions for checks and captures, creates a phenomenon that Berliner (1973, 1974) has called the *horizon effect*. Berliner's general observation is that, whenever search is terminated (short of the end of the game) and a static evaluation function is applied, the program's "reality exists in terms of the output of the static evaluation function, and anything that is not detectable at evaluation time does not exist as far as the program is concerned" (1974, p. I-1).

Two kinds of errors ensue. The first is called the negative horizon effect: The program manipulates the timing of moves to force certain positions to appear at the search horizon, and it thus may conclude that it has avoided some undesirable effect when in fact the effect has only been delayed to a point beyond the horizon. A second kind of error, the positive horizon effect, involves reaching for a desirable consequence: Either the program wrongly concludes that the consequence is achievable, or it fails to realize that the same consequence could also be achieved later in the game in a more effective form. This last problem, Berliner believes, can be met only by finding ways to represent and use more chess knowledge than traditional programs have included (1974, p. I-7).

For most of the errors coming from the horizon effect, however, the diagnosis is that the typical definitions of quiescence are highly oversimplified. Ideally a position would be considered quiescent only when the static evaluation function, applied to that position, could return a realistic value, that is, when the value of every term included in the function had become stable. A quiescence search that pursues only captures and checking moves, however, considers only changes in the material term. The material term itself, moreover, usually reflects only the presence of the pieces on the board; its value will be unchanged by a move that guarantees a capture later instead of making a capture now.

To get around the problems arising from inadequate quiescence analysis, a first approach called *secondary search* was developed by Greenblatt (1967): Whenever a move appeared, on the basis of the regular search (including quiescence), to be the best move considered so far, the predicted line of play was extended by searching another two ply (plus quiescence) to test the evaluation. Berliner points out, however: "The horizon effect cannot be dealt with adequately by merely shifting the horizon" (1974, p. I-4). One direction in current work, therefore, looks toward a much fuller quiescence analysis as a substitute for arbitrary depth bounds. (See Harris, 1975, 1977; Slate & Atkin, 1977, pp. 115-117; and, for an early example, Newell & Simon, 1972, pp. 678-698.) Berliner meanwhile is developing a general algorithm, not limited to chess, for causing tree search to terminate with a best move, even though no depth limit has been set and no full path to a win has been found (Berliner, 1977c, 1978b).

Iterative deepening. Despite its drawbacks, most current programs still use a fixed-depth search, extended for checks and capture sequences. A variation used by CHESS 4.5 (Slate & Atkin, 1977) is called *iterative deepening*: A complete search, investigating all legal moves (subject to alpha-beta pruning), is done to depth 2, returning a move. The search is then redone to depth 3, again to depth 4, and so on until a preset time limit is exceeded. For efficiency, information from earlier iterations is saved for use in later ones. Running on the very fast CDC Cyber 176, the program searches to an average depth of 6 plies in tournament play, with search trees averaging 500,000 nodes (Newborn, 1978). It is the first program to have achieved an Expert rating in human play. In the fall of 1978 a new version, CHESS 4.7, was reportedly rated 2160 (Levy, 1979); Master ratings begin at 2200. It remains an open question how much stronger the program can become.

Ordering of Search

The Shannon paradigm did not specify any particular order in which the nodes of the search tree were to be explored or in which moves from a given node were to be considered. For efficient use of space, the order of node expansion is usually depth-first; a depth-first algorithm needs to store explicitly only those nodes on the path it is currently investigating and not the parts of the tree where search has been completed.

With the invention of alpha-beta pruning, the order of considering moves within a depth-first search became highly significant. If the order is ideal, then in a tree with branching factor b the number of nodes that must be examined at depth d is reduced from b^d to only about $2b^{d/2}$. (See Article C5b.) For example, Shannon's estimated 10^9 chess positions at depth 6 would be reduced to around 50,000. It also follows that, for a constant number of tip nodes examined, correct ordering of the moves for alpha-beta cutoffs would allow the search depth to be roughly doubled. In general, the desired ordering is one in which the first move considered at a position is the best move for the player whose turn it is. Usually, of course, there is no method guaranteed to achieve this ordering, for if there were, it would enable moves to be chosen with no search at all. Several heuristics have been used, however, to try to approximate optimal ordering.

Perhaps the simplest idea for move ordering is the *fixed-ordering* method mentioned in Article C5b: For each move from a node, generate a new node for the resulting position, apply the static evaluation function to the position, and order the nodes according to this preliminary estimate. For greater efficiency, several programs have used a separate function for move ordering, which applies to the move itself instead of to the position that results from it (Greenblatt, 1967; Berliner, 1974, p. II-16; Adelson-Velskiy, 1975). In either case the game tree is explored by an *ordered depth-first search* (Article C3a).

A fuller basis for choosing which move to consider first is provided by Slate and Atkin's iterative deepening technique, which makes repeated depth-first searches. Each iteration constructs a line of play, down to its depth limit, consisting of apparently best moves. The following iteration, going one ply deeper, thus has available an estimated best move from each position along this line of play. (See Slate & Atkin, 1977, pp. 102-103.)

A further approach to move ordering makes explicit the idea of a *refutation move*: For each move that is not a best move, it should be shown as quickly as possible that the move is bad. To do this, strong replies should be considered first, which may refute the move proposed. Typical implementations consider all capturing moves first, and then consider *killer moves*. The idea here, called the *killer heuristic*, is that if a move has served as a refutation in some previously examined position that is similar to the current one, it is likely to be a refutation in the current position too. For more on the killer heuristic and other refutation techniques, see Slate and Atkin (1977), Adelson-Velskiy (1975), Gillogly (1972), and Frey (1977).

Once the moves have been ordered at a given node and the search has moved downward, following the move that seemed best, it may turn out that this move is actually a very bad one for reasons that were not apparent earlier. Since accurate move ordering is important to maximizing alpha-beta cutoffs, it might be worthwhile at this point to go back, reorder the moves, and start again with a different estimated best move. Such a procedure, called *dynamic ordering*, was investigated by Slagle and Dixon (1969), using the game of kalah. They reported a modest improvement over fixed ordering for trees of depth at least 6. On the other hand, Berliner's chess program experienced a serious increase in running time when dynamic ordering was used (1974, p. IV-14). A procedure somewhat similar to dynamic ordering was also used by Samuel (1967).

If dynamic ordering is carried to its limit, so that reordering is considered every time a node is expanded instead of only under more limited conditions, the search procedure in effect changes from depth-first to *best-first*. That is, the move considered next (or the position to which it leads) is on some estimate the most promising in the entire search tree generated so far, subject to whatever depth limit exists. Nilsson (1968, 1971) implements this idea by adapting his algorithm for best-first AND/OR tree search (C4) to game trees. Harris (1975, 1977) suggests another adaptation, in which the motivation of maximizing alpha-beta pruning no longer plays a role and instead the objective is to expand the most active positions first, using a thorough quiescence analysis rather than a depth limit as the criterion for search termination.

Width of Search

The techniques discussed so far are consistent with the idea that all legal moves from a position must be examined, at least sufficiently to establish that they can be safely pruned by alpha-beta. This consideration of all legal moves is referred to as *full-width searching*. Some of the earliest programs used a full-width search for simplicity; strong current programs use it because of the great difficulty in determining, without search, which moves can be safely ignored (Turing, 1953; Kister et al., 1957; Gillogly, 1972; Adelson-Velskiy et al., 1975; Slate & Atkin, 1977). The problem, of course, is that an excellent move may look very poor at first sight.

Yet the average number of legal moves from a chess position is at least 30, and even with a maximum of alpha-beta pruning the tree grows exponentially. Making the search more selective was Shannon's second requirement to change a program from Type A to Type B. Many people have been convinced that such selectivity is essential to a strong chess program, both in order to increase search depth and to permit more sophisticated evaluation of the nodes remaining in the search tree. Berliner, for example, has advocated reducing the total search tree size to at most 5000 nodes, with a branching factor of less than 1.9 (1974, p. I-16). Although some reconsideration of these ideas has been prompted by the success of CHESS 4.7 using full-width search, it appears that that program is still weak at long endgame sequences (see Berliner, 1978a; Michie & Bratko, 1978). Moreover, there are other games for which it is even clearer that full-width search is not the answer. For the game of *go*, for example, the average branching factor has been estimated at perhaps 200 (Thorp & Walden, 1970), and for backgammon, where legal moves depend on the throw of the dice as well as the board position, the factor is over 800 (Berliner, 1977a).

Various devices have been tried in the effort to increase the selectivity of the search without missing good moves. Some are conceptually simple, introducing little or no new chess-specific knowledge into the program. Others attempt to formulate and use chess concepts as sophisticated as those a chess master might employ. The remainder of this section reviews mainly the earlier search-controlling devices. The following section mentions work, some of which moves outside the Shannon paradigm, in which the effort to capture expert chess knowledge becomes primary.

Forward pruning. One way of limiting the number of moves to be considered introduces no new complications: Simply generate all legal moves at a position, use a fixed-ordering scheme to sort them according to their apparent goodness, or *plausibility*, and then discard all but the best few moves. Such a technique, called *plausible-move generation* or *forward pruning*, was used by Kotok (1962) and Greenblatt (1967); see also Samuel (1967). A further feature of these programs, sometimes called *tapered forward pruning*, was that the number of moves retained was a function of the depth at which they were generated. For example, Greenblatt's program in tournament play retained 15 moves from a position at either of the top two levels of the tree, 9 moves at the next two levels, and 7 moves thereafter. These figures could be increased in special cases--for example, to be sure that moves of more than a single piece were considered.

Another form of forward pruning, distinct from plausible move generation, operates not at the time when moves are originally generated but later, when one of these moves (or the position to which it leads) is being selected for further exploration. At this point a preliminary estimate of the value of the move or position may already have been made by the move-ordering scheme. If this estimate is outside the limits alpha and beta, the currently known bounds on the outcome of the entire search (see C5b), the node is pruned without further investigation. It is possible, of course, that the actual backed-up value of the node would have turned out to be between alpha and beta. In that case a good move may have been missed. (See Samuel, 1967; Berliner, 1974, p. IV-13.)

Still another basis for forward pruning has been explored by Adelson-Velskiy et al. (1975). They observe that KAISSA's search trees include many lines of play that a human would consider absurd, not necessarily because the moves are bad a priori, but because the human player has already considered and rejected the same moves in an analogous position. The proposal, then, is to remember moves that have been found to be absurd (on some definition) and to reject them in other positions too unless there has been an appropriate change of circumstances. In effect, this *method of analogies* involves trying to establish conditions under which a refutation is guaranteed to be effective. Then the line of play constituting the refutation would not need to be explored separately every time it is applicable. (See Frey, 1977, p. 68.)

Goal-directed move generation. Returning to the initial generation of moves, there is another kind of plausible move generator that comes closer to mimicking the way that humans might decide which moves are worth considering. Instead of generating all legal moves and discarding some, this approach does not generate moves at all unless they seem relevant to some *goal*. The earliest step in this direction was Bernstein's program (1959), which contained a sequence of board features to be tested for and a procedure for generating moves in response to each feature that was present. The first few tests in the sequence were (1) is the king in check? (2) can material be gained, lost, or exchanged? and (3) is castling possible? A maximum of 7 plausible moves was returned. Questions later in the sequence were not asked if earlier questions caused the maximum to be reached. Searching to a fixed depth of 4 ply, the program generated trees with about 2400 tip nodes.

More explicitly goal-directed move generation was included in Newell, Shaw, and Simon's 1958 chess program (Newell, Shaw, & Simon, 1963a; Newell & Simon, 1972). Indeed, the entire program was organized in terms of goals, although only three--material, center control, and piece development--were actually implemented. At each turn, the program began by making a preliminary analysis to decide which of the goals were relevant to the situation; these were entered, in order of importance, on a current goal-list. It was intended, in a more fully developed program, that as the game progressed the goals of center control and development would drop out, since they are important mainly in the opening, and would be replaced by others more appropriate to later phases of the game.

Each active goal in the Newell, Shaw, and Simon program was responsible for generating relevant moves at the first level of the tree. In addition, each goal contained its own separate generator for moves at deeper levels, its own criteria for whether a position was dead, and its own static evaluation function. The search proceeded, in a highly selective manner, until the tip nodes were dead with respect to all active goals. Static evaluations with respect to the various goals were combined lexicographically, so that the highest priority goal was dominant and the others served only as tiebreakers. Newell and Simon report that the program's average search tree contained only 13 nodes--with no apparent loss in playing power compared to other programs up to that time (1972, p. 694).

Knowledge-based Programs

The Bernstein and Newell, Shaw, and Simon programs were early efforts to introduce significant chess knowledge, organized in human terms, to limit brute-force search. The actual knowledge was very sketchy; apparently neither program ever won a game (see Newell & Simon, 1972, pp. 677, 690).

An attempt at fuller use of chess knowledge was made in Berliner's program, CAPS-II (1974, 1977b). Much of the work involved developing a representation suitable for use in selectively generating moves, making preliminary evaluations of the moves so proposed, and describing the actual consequences discovered when a move was tried. The moves generated depend on the current goal state, which may be King in Check, Aggressive, Preventive Defense, Nominal Defense, Dynamic Defense, or Strategy. In contrast to the Newell, Shaw, and Simon program, the goal states are mutually exclusive, and state transitions occur dynamically as the tree is searched, in accordance with a complex flowchart. An important feature of the program, the Causality Facility, relates to both move generation and move ordering, as well as to pruning in some cases. The problem it attacks is a general one in tree searching: When a path has been explored and found unsatisfactory, most programs have no way to diagnose what went wrong and use this information in deciding where to search next.

The basic search algorithm in CAPS-II is depth-first, with minimaxing and alpha-beta pruning. The Causality Facility operates as a refinement on this search. A first new feature is that, whenever a value is backed up in the search tree as a tentative minimax value, certain information is accumulated about the consequences of the move or moves that produced the value. The data structure in which the information is stored is called a Refutation Description. As the basis for making use of the Refutation Description, the program uses a variable representing the expected value of the position at the root of the search tree; this value, which may be updated during the search, lies somewhere between the bounds given by alpha and beta. Now, the tentative value newly backed up to a node can be compared with the expected value. If the comparison is unsatisfactory, the Causality Facility uses the Refutation Description to decide whether the last move tried from the node could have been responsible. It generates a list of alternative moves from the node, with the aim of avoiding the unsatisfactory result. These moves are compared with the list of moves from the node that had been generated earlier but which have not yet been tried. The comparison is used to reorder moves already on the untried list and, depending on the state the program is in, to add new moves to the list and to prune old ones.

Whereas Berliner's program plays the full game of chess, there are several other recent programs which, in their emphasis on representing chess knowledge, limit their task to solving problems that involve only selected aspects of the game. Two of these are the programs of Pitrat (1977) and Wilkins (1979). In each, the task is to find a line of play that wins material, beginning from a given middle-game position. The approach in both programs is to work backward from the goal of winning material to a structure of subgoals that constitutes a *plan*. (See Planning.) Wilkins's program, PARADISE, for example, has as a main theme the expression of chess concepts, like making a square safe for a piece or safely capturing a piece, in terms that can be used as subgoals and eventually reduced to specific moves. Initially, a plan is based not on search but on an extensive analysis of the originally given position; it may contain conditional branches depending on general categories of moves with which the opponent might reply. The general plan is then used to guide search, generating a very small tree. Moves considered for the program to make are only those relevant to the current subgoal; for the simulated opponent, all reasonable defensive moves are considered. If search shows that the plan has failed, a causality facility similar to Berliner's is used to analyze the difficulty and suggest a new plan.

Both the Pitrat and the Wilkins programs have succeeded in solving problems where the winning line of play goes to a depth of around 20 ply. Pitrat reports, for a set of 11 problems, that search tree sizes ranged from about 200 to 22,000 nodes; computation time varied from under 3 seconds to about 7.5 minutes. Wilkins's PARADISE generates smaller trees but uses more time; for 89 problems solved, the number of nodes in the search tree ran from a minimum of 3 to a maximum of 215, and time to find the solution varied from 19 seconds to 33 minutes. Wilkins also reports a good success rate compared to previous programs tested on the same set of problems, including Berliner's program, Gillogly's TECH, and an earlier version of CHESS 4.5. The programs other than PARADISE, however, were tested with a time limit of only 5 minutes per problem.

A final example of the use of chess knowledge to solve a class of problems is the work of Donald Michie and his colleagues on chess endgames (e.g., Bratko, Kopec, & Michie, 1978; Michie & Bratko, 1978). Here each combination of pieces with which the endgame may be played is treated as posing a separate problem. One problem, denoted KNKR, is to defend with king and knight against king and rook, starting from any of some 3 million legal positions involving only those pieces. The objective is to provide the program with enough knowledge about this specific class of chess problems to achieve theoretically correct play, even in situations where chess masters sometimes err, and to accomplish this using only a moderate amount of search.

The program's knowledge is encoded in a data structure called an Advice Table, within which patterns occurring on the board may be described. Each pattern has an associated list of goals, or "pieces of advice," in the order in which they should be attempted. The object then becomes to find a solution--in the sense of a solution subtree of an AND/OR tree (C2)--to the problem of satisfying one of the goals. Unlike a standard AND/OR tree search, however, the "advice" includes not only a definition of when tip nodes should be considered terminal, but also constraints that every intermediate node in the solution tree must satisfy.

The amount of search required to find a solution using an Advice Table depends on how much knowledge the table contains. If the only goal provided were avoidance of mate, a search to the impossible depth of 85 ply would be needed to find the best defense from some positions. With the additional advice not to lose the knight and to keep king and knight together, search to about 10 ply is sufficient. With the further refinements included in the actual Advice Table, the program is reported to play the KNKR endgame at master level using only a 4-ply search.

References

See Adelson-Velskiy, Arlazarov, & Donskoy (1975), Berliner (1973), Berliner (1974), Berliner (1977a), Berliner (1977b), Berliner (1977c), Berliner (1978a), Berliner (1978b), Bernstein et al. (1959), Bratko, Kopec, & Michie (1978), Charness (1977), Frey (1977), Gillogly (1972), Greenblatt, Eastlake, & Crocker (1967), Griffith (1974), Harris (1974), Harris (1975), Harris (1977), Hearst (1977), Kister et al. (1957), Kotok (1962), Levy (1979), Michie (1977), Michie & Bratko (1978), Mittman (1977), Newborn (1975), Newborn (1978), Newell, Shaw, & Simon (1963a), Newell & Simon (1972), Nilsson (1969), Nilsson (1971), Pitrat (1977), Samuel (1963), Samuel (1967), Shannon (1950), Slagle & Dixon (1969), Slagle & Dixon (1970), Slate & Atkin (1977), Thorp & Walden (1970), Turing (1953), and Wilkins (1979).

D. Example Search Programs

D1. Logic Theorist

The Logic Theorist (LT) was a program written by Allen Newell, J. C. Shaw, and H. A. Simon in 1956, as a joint project of the RAND Corporation and the Carnegie Institute of Technology. It was one of the earliest programs to investigate the use of heuristics in problem solving. The term *heuristics*, as used by Newell, Shaw and Simon, referred to "the complex processes . . . that are effective in problem-solving." They stated,

> We are not interested in methods that guarantee solutions, but which require vast amounts of computation. Rather, we wish to understand how a mathematician, for example, is able to prove a theorem even though he does not know when he starts how, or if, he is going to succeed. (Newell, Shaw, & Simon, 1963b, p. 109)

Heuristics were thus identified with processes "that may solve a given problem, but offer no guarantee of doing so" (p. 114; see also Overview).

In descriptions of the Logic Theorist program, the heuristics discussed by Newell, Shaw, and Simon relate principally to limiting the search space by means of an apt problem formulation. Within the defined space, the search was blind except for some minor selectivity in the selection of operators (see C3a).

The problem domain of the Logic Theorist is the proof of theorems in the propositional calculus (see Representation.Logic). The basis is Whitehead and Russell's **Principia Mathematica**, from which both axioms and theorems to be proved were taken. There are five axioms, as follows:

1. $(p \vee p) \supset p$
2. $p \supset (q \vee p)$
3. $(p \vee q) \supset (q \vee p)$
4. $[p \vee (q \vee r)] \supset [q \vee (p \vee r)]$
5. $(p \supset q) \supset [(r \vee p) \supset (r \vee q)]$

Some typical theorems that LT was given to prove include:

2.01. $(p \supset \sim p) \supset \sim p$
2.45. $\sim(p \vee q) \supset \sim p$
2.31. $[p \vee (q \vee r)] \supset [(p \vee q) \vee r]$

The numbering of the theorems is taken from Whitehead and Russell. In some cases, the data given the program included not only the axioms but also previously proved theorems from that work. When all earlier theorems were included with the axioms, the program succeeded in proving 38 of the first 52 theorems in Chapter 2 of **Principia Mathematica**, in the sequence given there.

The program operates by *reasoning backward*, from the theorem to be established, to the axioms and given theorems. Three operators were provided for reducing the theorem to be proved, let us say X, to an axiom or theorem. These operators were:

Detachment: To show X, find an axiom or theorem of the form A ⊃ X, and transform the problem to the problem of showing A.

Forward chaining: To show X where X has the form A ⊃ C, find an axiom or theorem of the form A ⊃ B, and transform the problem to the problem of showing B ⊃ C.

Backward chaining: To show X where X has the form A ⊃ C, find an axiom or theorem of the form B ⊃ C, and transform the problem to the problem of showing A ⊃ B.

Since the axioms and given theorems contain variables, consideration must be given to the means for deciding whether a problem has in fact been reduced to something known. The question is whether a current problem expression X is an instance of an axiom or known theorem. The test, called the Substitution Test, uses two rules of inference distinct from those reflected in the operators:

Substitution: A variable in a theorem may be replaced, in all its occurrences throughout the theorem, by an expression. For example, substituting the expression "p v q" for the variable "p" transforms

$$p \supset (q \vee p)$$

into

$$(p \vee q) \supset [q \vee (p \vee q)].$$

Replacement: The connective "⊃" is interchangeable with its definition. That is, if p and q are expressions, then

$$p \supset q$$

can be replaced by

$$\sim p \vee q$$

and vice versa.

As well as being used to determine whether a proof is complete, the substitution test is also essential for determining what applications of the three operators are possible with respect to a given problem expression.

The general algorithm used by the Logic Theorist is a blind, breadth-first state-space search using backward reasoning. The initial state corresponds to the original theorem to be proved. To test whether an expression has been proved, the program applies the substitution test, pairing the problem expression with each axiom and assumed theorem, in turn. If substitution fails, the expression is placed on a list of open problems; problems are selected from this list to become the current problem in first-in, first-out order.

To a problem selected from the list, each of the three operators is applied, in fixed order and in all possible ways, to generate new open problems. The search terminates with success as soon as a single problem is generated that passes the substitution test, since this means that a path has been completed between an axiom and the original problem. The search fails if it exceeds time or space limits, or if it runs out of open problems.

An example of a case in which the latter occurs is the attempted proof of the theorem p or ~~~p .

To succeed with this proof, LT would have needed more powerful operators; this particular problem required the ability, which LT lacked, to transform a problem to a set of subproblems, or conjunctive subgoals, which *all* had to be solved in order to solve the original problem.

There are some qualifications to the preceding general description of LT. One concerns the statement that each operator is applied to the current problem in every possible way, that is, that the current problem expression is matched against every axiom and assumed theorem to determine the applicability of any of the operators to that expression-axiom pair. In fact, the program attempted a match for the purpose of discovering an appropriate substitution only if the pair had passed a test indicating equality of certain gross features, such as the number of distinct variables in each. This test for similarity occasionally rejected a pair for which a substitution in fact would have been possible, thus excluding a proof the program would otherwise have found. Overall, the utility of this similarity test was considered rather marginal.

Some other additions, apparently made in a later version of the program (see Newell & Simon, 1972, pp. 125-128), included (a) ordering the open problems, taking up those involving simpler expressions first instead of proceeding in a strictly breadth-first order, and (b) rejecting some subproblems entirely as too complicated or apparently unprovable. In the implementation of these features, the latter appeared to be the more effective measure in reducing search effort. There was also experimentation, as mentioned previously, with the number of theorems that could be assumed as *given* in addition to the basic axioms. The conclusion on this point was that "a problem solver may be encumbered by too much information, just as he may be handicapped by too little" (Newell & Simon, 1972, p. 127).

References

See Newell, Shaw, & Simon (1963b), Newell & Simon (1972), and Whitehead & Russell (1925).

D2. General Problem Solver

The General Problem Solver (GPS) was developed by Allen Newell, J. C. Shaw, and H. A. Simon beginning in 1957. The research had a dual intention: It was aimed both at getting machines to solve problems requiring intelligence and at developing a theory of how human beings solve such problems. GPS was the successor of the authors' earlier Logic Theorist program (Article D1), whose methods had only a slight resemblance to those used by humans working on similar problems. Development of GPS continued through at least ten years and numerous versions of the program. The final version, described in detail in Ernst and Newell (1969), was concerned with extending the generality of the program, not with the psychological theory.

The name "General Problem Solver" came from the fact that GPS was the first problem-solving program to separate its general problem-solving methods from knowledge specific to the type of task at hand. That is, the problem-solving part of the system gave no information about the kind of task being worked on; task-dependent knowledge was collected in data structures forming a *task environment*. Among the data structures were *objects* and *operators* for transforming objects. A task was normally given to GPS as an initial object and a desired object, into which the initial object was to be transformed. GPS objects and operators were similar to the states and operators of a state-space problem representation (Article B1).

The general problem-solving technique introduced by GPS, however, does not fit neatly into either the state-space or the problem-reduction representation formalisms. It differs from a standard state-space search (e.g., Article IIC1) in the way it decides what path to [illegible] next. This technique, called *means-ends analysis*, is a major theoretical contribution of the program. It assumes that the *differences* between a current object and a desired object can be defined and classified into types and that the operators can be classified according to the kinds of difference they might reduce. At each stage, GPS selects a single relevant operator to try to apply to the current object. The search for a successful operator sequence proceeds depth first as long as the chosen operators are applicable and the path shows promise. Backup is possible if the current path becomes unpromising--for example, if eliminating one difference has introduced a new one that is harder to get rid of.

An important feature of means-ends analysis is the fact that the operator selected as relevant to reducing a difference may in fact be inapplicable to the current object. Rather than rejecting the operator for this reason, GPS attempts to change the current object into an object appropriate as input to the chosen operator. The result of this strategy is a recursive, goal-directed program that records the search history in an AND/OR graph (Article B2) with partial development of nodes (Article C3a).

Goals and Methods

The most important data structure used by GPS is the *goal*. The goal is an encoding of the current situation (an object or list of objects), the desired situation, and a history of the attempts so far to change the current situation into the desired one. Three main types of goals are provided:

1. **Transform** object A into object B.
2. **Reduce** a difference between object A and object B by modifying object A.
3. **Apply** operator Q to object A.

Associated with the goal types are *methods*, or procedures, for achieving them. These methods, shown in a simplified version in Figure 1, can be understood as problem-reduction operators that give rise either to AND nodes, in the case of **transform** or **apply**, or to OR nodes in the case of a **reduce** goal. The initial task presented to GPS is represented as a **transform** goal, in which A is the initial object and B the desired object.

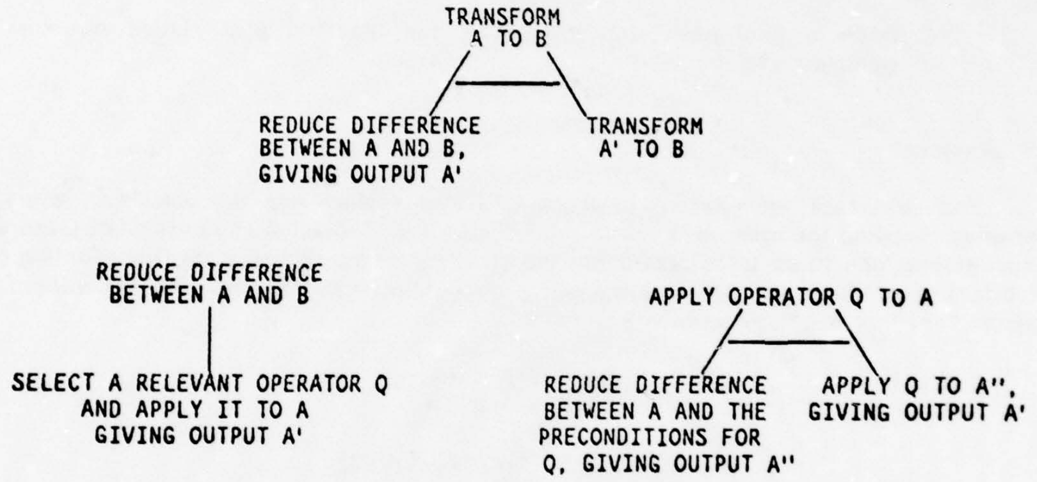

TRANSFORM A TO B
REDUCE DIFFERENCE BETWEEN A AND B, GIVING OUTPUT A'
TRANSFORM A' TO B
REDUCE DIFFERENCE BETWEEN A AND B
SELECT A RELEVANT OPERATOR Q AND APPLY IT TO A GIVING OUTPUT A'
APPLY OPERATOR Q TO A
REDUCE DIFFERENCE BETWEEN A AND THE PRECONDITIONS FOR Q, GIVING OUTPUT A''
APPLY Q TO A'', GIVING OUTPUT A'

Figure 1. The three GPS methods for problem reduction.

The recursion stops if the goal is primitive--that is, if for a **transform** goal there is no difference between A and B; and if for an apply goal the operator Q is immediately applicable. For a **reduce** goal, the recursion may stop, with failure, when all relevant operators have been tried and have failed.

Selection of Operators

In trying to transform object A to object B, the **transform** method uses a matching process to discover the differences between the two objects. The possible types of difference are predefined and ordered by estimated difficulty, for each kind of task. The most difficult difference found is the one chosen for reduction. A domain-dependent data structure called the *Table of Connections* lists the operators relevant to reducing each difference type.

Depth Bounds

Several heuristics are provided to prevent GPS from following a false path indefinitely. Some of the bases on which a current goal may be abandoned, at least temporarily, are the following:

1. Each goal should be easier than its parent goal.

2. Of a pair of AND nodes representing subgoals generated by **transform** or **apply**, the second subgoal attempted should be easier than the first.

3. A newly generated object should not be much larger than the objects occurring in the topmost goal.

4. Once a goal has been generated, the identical goal should not be generated again.

An Example

The first task environment to which GPS was applied was the domain of the Logic Theorist: proving theorems in the propositional calculus. The initial and desired objects were expressions, one to be transformed into the other by means of operators representing rules of inference. There were twelve operators altogether, including the following rules. (The symbol "==>" means "may be rewritten as.")

Rule 1. $A \vee B$ ==> $B \vee A$
$A \wedge B$ ==> $B \wedge A$

. . .

Rule 5. $A \vee B$ <==> $\sim(\sim A \wedge \sim B)$

Rule 6. $A \supset B$ <==> $\sim A \vee B$

Six possible difference types were recognized:

(a) occurrence of a variable in one expression but not the other,
(b) occurrence of a variable a different number of times in the two expressions,
(c) difference in sign,
(d) difference in binary connective,
(e) difference in grouping, and
(f) difference in position of components.

The list just given is in decreasing order of assumed difficulty. Every difference between main expressions, however, was considered more difficult than any difference between subexpressions.

With this background, a trace (slightly simplified) of GPS's performance on a simple example can be given. The problem is to transform the initial expression

$$R \wedge (\sim P \supset Q) ,$$

denoted L1, into the desired expression

$$(Q \vee P) \wedge R ,$$

denoted L0. The trace is shown below.

```
Goal 1: Transform L1 into L0.
    Goal 2: Reduce positional difference between L1 and L0.
        Goal 3: Apply Rule 1 to L1.

        Return L2: $(\sim P \supset Q) \wedge R$

    Goal 4: Transform L2 into L0.
        Goal 5: Reduce difference in connective between
                left subexpressions of L2 and L0.

            Goal 6: Apply Rule 5 to left part of L2.

                Goal 7: Reduce difference in connective
                        between left part of L2 and
                        precondition for Rule 5.
                Reject goal 7 as no easier than goal 5.

            Goal 8: Apply Rule 6 to left part of L2.

            Return L3: $(P \vee Q) \wedge R$

        Goal 9: Transform L3 into L0.

            Goal 10: Reduce positional difference
                     between left parts of L3 and L0.

                Goal 11: Apply Rule 1 to left part of L3.

                Return L4: $(Q \vee P) \wedge R$

            Goal 12: Transform L4 to L0.

            No difference exists, so problem is solved.
```

The Problem of Generality

GPS was intended to model generality in problem solving through use of the broadly applicable techniques of heuristic search, and the strategy of means-ends analysis in particular. The implementation of these techniques was dependent on the internal representation of objects and operators. These representations, in early versions of GPS, were nicely suited to logic tasks like the example above. But they were inadequate for many other kinds of heuristic search problems. Before Ernst's extensions to the program (Ernst & Newell, 1969), GPS had in fact solved only two problems outside the logic domain.

The object of Ernst's work was to extend the number of kinds of problems that GPS could handle while holding its power at a constant level. One of his generalizations was in the representation of objects. Earlier, a desired object had had to be specified by giving its exact form. Forms containing variables and lists of forms could be used if necessary. But these too were inadequate for representing symbolic integration problems, in which the desired object is any form whatever that does not contain an integral sign. Hence the description of a desired object by a list of constraints was introduced.

Another change was in the representation of operators, originally specified by giving the form of the input object and the form of the resulting output object. For some kinds of problems, it was desirable to have other tests of applicability besides the form of the input object, and to be able to describe the output object as a function of the input. A third change enabled GPS to deal with unordered sets of symbols, eliminating the need for special operators to permute their elements.

The generalized program succeeded in solving problems of 11 different kinds including symbolic integration, resolution theorem proving, and a variety of puzzles. Each generalization, however, entailed changes in the ways the problem representations could be processed, and these led in turn to deterioriation with respect to the kinds of differences that could be detected. The only representable differences became "local" ones. An example of a global difference, which GPS could no longer recognize, was the total number of times a variable occurred in a logic formula. Consequently, theorem proving in the propositional calculus was not among the eleven tasks that the final version of GPS could do.

In the task domains in which GPS did succeed, it could solve only simple problems; and those, less efficiently than special-purpose problem solvers. If a long search was required, it ran out of memory space; and even easy problems, if they needed objects as complex as a chess position, quickly exhausted memory on a machine with 65K words. But GPS was not expected to be a performance program. What it yielded, in its authors' view, was "a series of lessons that give a more perfect view of the nature of problem solving and what is required to construct processes that accomplish it" (Ernst & Newell, 1969, p. 2). Although additional generalizations, such as game playing, were considered feasible, the authors concluded that GPS needed no further programming accretions and recommended that it be laid to rest.

References

See Ernst & Newell (1969), Newell & Ernst (1965), Newell, Shaw, & Simon (1960), Newell & Simon (1963), and Newell & Simon (1972).

D3. Gelernter's Geometry Theorem-proving Machine

Herbert Gelernter's geometry theorem-proving machine was a program written in 1959 at the IBM Research Center in New York. The program was written in an extended FORTRAN, the FORTRAN List Processing Language, and implemented on an IBM 704 computer. The purpose of the program was to solve problems taken from high-school textbooks and final examinations in plane geometry. As with Newell, Shaw, and Simon's Logic Theorist, which proved theorems in the propositional calculus, the fact that there were algorithms for solving problems in these domains was considered irrelevant, since the object was to explore the use of heuristic methods in problem solving.

The formal system within which the geometry program worked contained axioms on parallel lines, congruence, and equality of segments and angles. This set of axioms, which was not meant to be either complete or nonredundant, was along the lines of an elementary textbook. The axioms played the role of *problem-reduction* operators. Some examples are: (a) To show that two line segments are equal, show that they are corresponding elements of congruent triangles; (b) to show that two angles are equal, show that they are both right angles; and (c) to show that two triangles are congruent, show the equality of a side and two angles in corresponding positions, or of an angle and two sides. The operators for establishing congruence split the problem into three subproblems, each to be solved separately by showing equality for one pair of elements. Newell and Simon (1972, p. 138) indicate that the geometry machine was the first program that was able to handle *conjunctive subgoals*. The program works backwards from the theorem to be proved, recording its progress in what amounted to an AND/OR tree (Article B2).

Some examples of problems solved by the program were the following:

1. Given that angle ABD equals angle DBC, that segment AD is perpendicular to segment AB, and that segment DC is perpendicular to segment BC, show that AD equals CD.

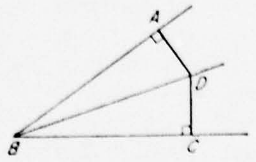


Figure 1. Diagram for problem 1.

2. Given that ABCD is a quadrilateral, with segment BC parallel to segment AD and with BC equal to AD, show that segment AB equals segment CD.

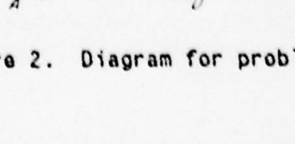


Figure 2. Diagram for problem 2.

A problem was given to the program in the form of a statement describing the premises and the goal. A proof was a sequence of statements giving the reduction of the goal to trivial goals--ordinarily, goals to establish an already established formula. One feature used to reduce the search effort needed to find a proof was the recognition of *syntactic symmetry*. Some examples of symmetric pairs of goals are the following:

a. If d(x,y) is the distance from point x to point y, then d(A,B) = d(C,D) is symmetric with d(D,C) = d(A,B).

b. If ABCD is a parallelogram and point E is the intersection of its diagonals AC and BD, then d(A,E) = d(E,C) is symmetric with d(B,E) = d(E,D).

The recognition of symmetry was used in two ways. First, if a given goal was ever reduced to a subgoal symmetric with it, the subgoal could be rejected as representing circular reasoning. Second, if parallel goals A and B were syntactically symmetric and goal A had been established, then goal B could be established by symmetry--in effect by saying, for the second half of the proof, "Similarly, B."

The most notable feature of the program, however, was an additional part of the problem statement used to avoid attempting proofs by blind syntactic manipulation alone. This input was a diagram, similar to Figures 1 and 2 (although specified by lists of coordinates), of the points and line segments mentioned in the theorem. The particular input figure was chosen to avoid spurious coincidences and reflect the greatest possible generality. Whenever a subgoal was generated, it was checked for consistency with the diagram. If false in the diagram, the subgoal could not possibly be a theorem and therefore could be *pruned* from the search tree. A slight qualification is that finite precision arithmetic, applied to the diagram, occasionally caused a provable subgoal to be pruned erroneously; but it was reported that the program had found other paths to the solution in such cases. It was estimated that the use of a diagram, together with the discard of subgoals representing circular reasoning, eliminated about 995 out of every thousand subgoals.

The diagram also served a second purpose: It provided an additional criterion by which a problem could be considered *primitive*. For example, a rigorous proof of the theorem in problem 1 would require showing that DB is a line segment and that BCD and BAD are triangles. The axioms needed would have been (a) if X and Y are distinct points, then XY is a line segment; and (b) if X, Y, and Z are three distinct non-collinear points, then XYZ is a triangle. For a limited class of such properties, the program did not require formal proof but rather considered them established if they were true in the diagram. It recorded explicitly the assumptions that had been made based on the diagram.

The central loop of the program repeatedly selects the next goal to try. Two heuristics were included for goal selection. One gave highest priority to classes of goals, such as identities, that could usually be established in one step. The second assigned a "distance" between the goal statement and the set of premise statements; after the one-step goals had been developed, the remaining goals were selected in order of increasing distance from the premise set.

Once a goal was chosen for development, the action taken depended on its status. Ordinarily, it would be reduced to subgoals and the subgoals, if consistent with the diagram but not sufficient to establish the current goal immediately, would be added to the list of goals to try. If no new acceptable subgoals were generated, the program checked whether a *construction* was possible--a construction being the addition to the premises of the problem of a line segment between two existing but previously unconnected points. The new segment would be extended to its intersections with other segments in the figure. New points could be added to the premises only if generated by such intersections.

A goal for which a construction was found possible was saved--to be tried again later if all goals not requiring construction should be exhausted. If the goal was later selected for a second try, a construction would be made and the problem started over with an expanded premise set. An example of the use of this technique occurs in problem 2, where in considering the goal AB = CD, the program generated a subgoal of showing that triangles ABD and CDB were congruent. The subgoal makes sense only if a line segment BD exists, so the segment is constructed, and the proof eventually succeeds.

References

See Elcock (1977), Gelernter (1959), Gelernter (1963), Gelernter, Hansen, & Gerberich (1960), Gelernter, Hansen, & Loveland (1963), Gelernter & Rochester (1958), and Gilmore (1970).

D4. Symbolic Integration Programs

Slagle's SAINT

James Slagle's SAINT program (Symbolic Automatic INTegrator) was written as a 1961 doctoral dissertation at MIT. The program solves elementary symbolic integration problems--mainly indefinite integration--at about the level of a good college freshman. SAINT was written in LISP and run interpretively on the IBM 7090.

The kinds of questions Slagle intended his thesis to address were some of the earliest questions for AI. They included, for example, "Can a computer recognize the kinds of patterns that occur in symbolic expressions? Just how important is pattern recognition? . . . Can intelligent problem solving behavior really be manifested by a machine?" (Slagle, 1961, p. 9). The domain of symbolic integration was chosen as a source of well-defined, familiar, but nontrivial problems requiring the manipulation of symbolic rather than numerical expressions.

The integration problems that SAINT could handle could have only elementary functions as integrands. These functions were defined recursively to comprise the following:

1. Constant functions.
2. The identity function.
3. Sum, product, and power of elementary functions.
4. Trigonometric, logarithmic, and inverse trigonometric functions of elementary functions.

Three kinds of operations on an integrand were available:

1. Recognize the integrand as an instance of a standard form, thus obtaining the result immediately by substitution. Twenty-six such standard forms were used. A typical one indicated that if the integrand has the form $c^v\,dv$, the form of the solution is $(c^v)/(\ln c)$.

2. Apply an "algorithm-like transformation" to the integral--that is, a transformation that is almost guaranteed to be helpful whenever it can be applied. Eight such transformations were provided, including (a) factoring out a constant and (b) decomposing the integral of a sum into a sum of integrals.

3. Apply a "heuristic transformation"--that is, a transformation carrying significant risk such that, although applicable, it might not be the best next step. The 10 heuristic transformations included certain substitutions and the technique of integration by parts. One technique that was not implemented was the method of partial fractions.

The program starts with the original problem as a goal, specified as an integrand and a variable of integration. For any particular goal, the strategy is first to try for an immediate solution by substitution into a standard form; failing that, to transform it by any applicable algorithm-like transformation; and finally to apply each applicable heuristic transformation in turn. Both the algorithm-like and the heuristic transformations, however, generate new goals, to which the same strategy may be applied. The result is an AND/OR graph of goals (Article B2).

The order in which goals are pursued by SAINT depends heavily on what operations can be applied to them. At the level of heuristic transformations, the algorithm is an *ordered search*: A list, called the Heuristic Goal List, keeps track of goals on which progress can be made only by applying heuristic transformations--that is, integrands that are not of standard form nor amenable to any algorithm-like transformation. To each goal on this list is attached an estimate of the difficulty of achieving it. The measure of difficulty used is the maximum level of function composition in the integrand. Other characteristics of the goal, such as whether it is a rational function, an algebraic function, a rational function of sines and cosines, and the like, are also stored as an aid to determining which heuristic transformations will in fact apply. The outer loop of the program repeatedly selects the goal that looks the easiest from the Heuristic Goal List, expands it by applying all applicable heuristic transformations, and possibly, as a result of the expansion, adds new elements to the Heuristic Goal List. The program terminates with failure if it runs out of heuristic goals to work on or if it exceeds a pre-set amount of working space.

An important qualification to this process concerns the use of standard forms and algorithm-like transformations. As soon as any new goal is generated (or the original goal read in), an immediate solution of it is attempted. The attempt consists of, first, checking whether the integrand is a standard form; if it is not, checking whether an algorithm-like transformation applies; and if one does, applying it and calling the immediate solution procedure recursively on each goal resulting from that transformation. When the recursion terminates, either the generated goal has been achieved or there is a set of goals--the generated goal itself or some of its subgoals--to be added to the Heuristic Goal List. During expansion of a node (one iteration of the outer loop), new heuristic goals are accumulated in a temporary goal list; only after expansion is complete are their characteristics computed and the additions made to the Heuristic Goal List.

Whenever a goal is achieved, the implications of its achievement are immediately checked. If it is the original goal, the program terminates successfully. Otherwise, if it was achieved by substitution into a standard form, it may cause the achievement of one or more parent goals as well. If it was achieved by solution of a sufficient number of its subproblems, it may not only cause its parent or parents to be achieved in turn, but may also make others of its subproblems, which have not yet been solved, superfluous. These checks are implemented in a recursive process, referred to as "pruning the goal tree," that is initiated as soon as any goal is achieved. Thus a heuristic goal can be achieved without having been fully expanded.

Moses's SIN

A second important symbolic integration program, SIN (Symbolic INtegrator), was written by Joel Moses in 1969, also as a doctoral dissertation at MIT. Its motivation and its strategy as an AI effort were quite different from those of SAINT. Whereas Slagle had compared the behavior of SAINT to that of freshman calculus students, Moses aimed at behavior comparable to expert human performance. He viewed SAINT as emphasizing generality in that it examined mechanisms, like heuristic search, that are useful in many diverse problem domains. SIN, in contrast, was to emphasize expertise in a particular, complex domain. To do this, it concentrated on problem analysis, using more knowledge about integration, than SAINT had employed, to minimize the need for search. In fact, Moses did not view SIN as a heuristic search program. Hence, the program will be described only briefly here; and a second part of Moses's dissertation, a differential equation solver called SOLDIER, will not be described.

SIN worked in three stages, each stage being capable of solving harder problems than the stage before. Stage 1 corresponded roughly to Slagle's immediate solution procedure but was more powerful. It used a table of standard forms; two of Slagle's algorithm-like transformations; and, most importantly, a method similar to one of Slagle's heuristic transformations, referred to as the Derivative-divides method. The idea behind this grouping of methods was that they alone would be sufficient to solve the most commonly occurring problems, without invoking the computationally more expensive machinery of the later stages.

A problem that stage 1 could not solve was passed on to stage 2. This stage consisted of a central routine, called FORM, and 11 highly specific methods of integration. (One of these methods was a program for integrating rational functions that had been written by Manove, Bloom, and Engelman, of the MITRE Corporation, in 1964.) In general, the task of FORM was to form a hypothesis, usually based on local clues in the integrand, about which method, if any, was applicable to the problem. Only rarely did more than one method apply. The routine chosen first tried to verify its applicability; if it could not, it returned to let FORM try again. If the routine did verify the hypothesis, however, SIN then became committed to solving the problem by that method or not at all. The method chosen either solved the problem using mechanisms internal to it or transformed the problem and called SIN recursively to solve the transformed problem.

Stage 3 of SIN was invoked, as a last resort, only if no stage 2 method was applicable. Two general methods were programmed here. One method was integration-by-parts, which used blind search, subject to certain constraints, to find the appropriate way to factor the integrand. The other was a nontraditional method based on the Liouville theory of integration and called the EDGE (EDucated GuEss) heuristic. This method involved guessing the form of the integral. The EDGE heuristic was characterized as using a technique similar to *means-ends analysis*, if its guess did not lead directly to a solution.

Performance of SAINT and SIN

SAINT was tested on a total of 86 problems, 54 of them chosen from MIT final examinations in freshman calculus. It succeeded in solving all but two. The most difficult problem it solved, both in terms of time and the number of heuristic transformations occurring in the solution tree (four), was the integral of

$$\frac{(\sec t)^2}{1 + (\sec t)^2 - 3(\tan t)}\ dt .$$

Slagle proposed additional transformations that would have handled the two failures, which were the integrals of

$$x(1+x)^{1/2}\,dx \quad \text{and} \quad \cos(x^{1/2})\,dx .$$

SIN, in contrast, was intended to model the behavior of an expert human integrator. The results of running SIN on all of Slagle's test problems were that more than half were solved in stage 1, and all but two of the rest (both of which used integration by parts) were solved in stage 2. After adjusting for the facts that SAINT and SIN ran on different machines and that one was interpreted and the other compiled, and for other factors making the programs difficult to compare, Moses estimated that SIN would run on the average about three times faster than SAINT. Taking into account a test on more difficult problems as well, he expressed the opinion that SIN was "capable of solving integration problems as difficult as ones found in the largest tables" (p. 140) and that it was fast and powerful enough for use in "a practical on-line algebraic manipulation system" (p. 6). For later developments in this direction, see Applications.Macsyma.

References

See Manove, Bloom, & Engelman (1968), Moses (1967), Slagle (1961), and Slagle (1963).

D5. STRIPS

STRIPS is a problem-solving program written by Richard Fikes and Nils Nilsson (1971) at SRI International. Each problem for STRIPS is a goal to be achieved by a robot operating in a simple world of rooms, doors, and boxes. The solution is a sequence of operators, called a *plan*, for achieving the goal. (For a review of the various senses of the word *plan*, see Planning). The robot's actual execution of the plan is carried out by a separate program, distinct from STRIPS. A later (1972) addition to the basic STRIPS system permits plans to be generalized and used again, giving the system some capacity for *learning*.

The Basic STRIPS System

The world model. The world in which the STRIPS robot works consists of several rooms connected by doors, along with some boxes and other objects that the robot can manipulate. STRIPS represents this world by a set of well-formed formulas in the first-order predicate calculus (see Representation.Logic). Some formulas in the world model are static facts, such as which objects are pushable and which rooms are connected. Other facts, such as the current location of objects, must be changed to reflect the actions of the robot.

Operators. The primitive actions available to the robot are represented by *operators*. Typical operators include going somewhere and pushing an object somewhere, the locations being given as parameters. Each operator has *preconditions* to its applicability; to push a box, for example, the robot must first be next to the box. The application of an operator is realized by making changes in the world model. The appropriate changes are given by a *delete list* and an *add list*, specifying the formulas to be removed from and added to the world model as a result of the operation. Thus, each operator explicitly describes what it changes in the world model.

A typical operator is GOTOB, which denotes the robot's going up to an object in the same room:

```
GOTOB (bx)  "go to object bx"
   Preconditions:  TYPE(bx,OBJECT) and
         THERE EXISTS (rx) [INROOM(bx,rx) and INROOM(ROBOT,rx)]
   Delete list:  AT(ROBOT,*,*), NEXTTO(ROBOT,*)
   Add list:  NEXTTO(ROBOT,bx)  .
```

The precondition statement requires that bx be an object and that both bx and the robot be in the same room, rx. The asterisks in the delete list represent arguments with any values whatever.

Method of operation. STRIPS operates by searching a space of world models to find one in which the given goal is achieved. It uses a *state-space representation* in which each state is a pair (world model, list of goals to be achieved). The initial state is (MO, (GO)), where MO is the initial world model and GO the given goal. A terminal state gives a world model in which no unsatisfied goals remain.

Given a goal G (stated as a formula in the predicate calculus), STRIPS first tries to prove that G is satisfied by the current world model. To do this, the program uses a modified version of the resolution-based theorem prover QA3 (Garvey & Kling, 1969). Typically the proof fails, within a pre-specified resource limit, because no more resolvents can be formed (see Theorem Proving.Resolution). At this point, STRIPS needs to find a different world model which the robot can achieve and which satisfies the goal. Because this task is complicated for a simple theorem prover, the system switches to a *means-ends analysis* similar to that of GPS (Article D2).

To do the means-ends analysis, the program extracts a *difference* between the goal and the current model and selects a "relevant" operator to reduce the difference. The difference consists of any formulas from the goal that remain outstanding when the proof attempt is abandoned (pruned, if this set is large). A relevant operator is one whose add list contains formulas that would remove some part of the difference, thereby allowing the proof to continue.

If the operator is applicable, the program applies it and tries to achieve the goal in the resulting model; otherwise, the operator's precondition becomes a new subgoal to be achieved. Since there may be several relevant operators at each step, this procedure generates a tree of models and subgoals. STRIPS uses a number of heuristics to control the search through this tree.

An Example of the Basic System's Performance

As a simple example, suppose the robot is in ROOM1 and the goal is for it to be next to BOX1, which is in adjacent ROOM2. The initial world model MO contains such clauses as

```
INROOM (ROBOT,ROOM1),
INROOM (BOX1,ROOM2),
TYPE (BOX1,OBJECT),
CONNECTS (DOOR12,ROOM1,ROOM2),
STATUS (DOOR12,OPEN), . . .
```

and the goal GO is

```
G0 = NEXTTO (ROBOT,BOX1)  .
```

GO is not satisfied, and the difference between it and the initial model is ~NEXTTO (ROBOT,BOX1). STRIPS determines that GOTOB (bx), defined above, is a relevant operator, with bx instantiated as BOX1. The operator instance GOTOB (BOX1), denoted OP1, is not immediately applicable (because the robot is in the wrong room), so its precondition G1,

```
G1 = TYPE (BOX1,OBJECT) and
     THERE EXISTS (rx) [INROOM (BOX1,rx) and INROOM (ROBOT,rx)]
```

becomes a new subgoal. Relevant operators for reducing the difference between G1 and the initial model MO are: OP2 = GOTHRUDOOR (dx,ROOM2) and OP3 = PUSHTHRUDOOR (BOX1,dx,ROOM1) (i.e., move the robot to the room with the box, or move the box to the room with the robot). If the former course (the better one, obviously) is selected, the precondition

```
G2 = STATUS (dx,OPEN)  and  NEXTTO (ROBOT,dx)  and
     THERE EXISTS (rx) [INROOM (ROBOT,rx) and CONNECTS (dx,rx,ROOM2)]
```

is the new subgoal. The difference ~NEXTTO (ROBOT,DOOR12) can be reduced by the operator OP4 = GOTODOOR (DOOR12), which is applicable immediately. Applying OP4 adds the clause NEXTTO (ROBOT,DOOR12) to the model, creating a new world model M1. G2 is now satisfied with dx = DOOR12, so OP2 can be instantiated as GOTHRUDOOR (DOOR12,ROOM2) and applied. This deletes the clause INROOM (ROBOT,ROOM1) and adds INROOM (ROBOT,ROOM2). G1 is now satisfied, so OP1 is applied, deleting NEXTTO (ROBOT,DOOR12) and adding NEXTTO (ROBOT,BOX1), the desired goal. The final plan is thus:

OP4:	GOTODOOR (DOOR12)
OP2:	GOTHRUDOOR (DOOR12,ROOM2)
OP1:	GOTOB (BOX1)

The corresponding solution path through the state space tree is as follows:

```
(M0, (G0))
    \
     (M0, (G1, G0))
           \
            (M0, (G2, G1, G0))
                      \
                       \ OP4
                        \
                  (M1, (G1, G0))
                          \
                           \ OP2
                            \
                        (M2, (G0))
                                \
                                 \ OP1
                                  \
                              (M3, ())
```

Generalization of Plans

In the basic STRIPS system, each new problem was solved from scratch. Even if the system had produced a plan for solving a similar problem previously, it was not able to make any use of this fact. A later version of STRIPS provides for generalizing plans and saving them, to assist both in the solution of subsequent problems and also in the intelligent monitoring of the robot's execution of the particular plan.

Triangle tables. A specific plan to be generalized, say (OP1, OP2, ..., OPn), is first stored in a data structure called a *triangle table*. This is a lower triangular array representing the preconditions for and effects of each operator in the plan. Some of its properties are the following:

1. Cell (i,0) contains clauses from the original model that are still true when operator i is to be applied and that are preconditions for operator i, OPi.

2. Marked (starred) clauses elsewhere in row i are preconditions for operator i added to the model by previous operators.

3. The effects of applying operator i are shown in row i+1. The operator's add list appears in cell (i+1, i). For each previous operator, say operator j, clauses added by operator j and not yet deleted are copied into cell (i+1,j).

4. The add list for a sequence of operators 1 through i, taken as a whole, is given by the clauses in row i+1 (excluding column 0).

5. The preconditions for a sequence of operators i through n, taken as a whole, are given by the marked clauses in the rectangular sub-array containing row i and cell (n+1, 0). This rectangle is called the i-th *kernel* of the plan.

The triangle table for the previous example is shown below. Operators have been renumbered in the order of their use.

1	*INROOM(ROBOT, ROOM1) *CONNECTS(D12, ROOM1,ROOM2)	OP1 GOTODOOR(D12)		
2	*INROOM(ROBOT, ROOM1) *CONNECTS(D12, ROOM1,ROOM2) *STATUS (D12, OPEN)	*NEXTTO (ROBOT,D12)	OP2 GOTHRUDOOR (D12,ROOM2)	
3	*INROOM(BOX1, ROOM2) *TYPE(BOX1, OBJECT)	NEXTTO (ROBOT,D12)	*INROOM (ROBOT,ROOM2)	OP3 GOTOB(BOX1)
4			INROOM (ROBOT,ROOM2)	NEXTTO (ROBOT,BOX1)
	0	1	2	3

Figure 1. A triangle table.

Method of Generalization. The plan is generalized by replacing all constants in each of the clauses in column 0 by distinct parameters and the rest of the table with clauses that assume that no argument to an operator has been instantiated. The result may be too general, so the proof of the preconditions for each operator is run again, noting any substitutions for parameters that constrain the generality of the plan. Some further corrections are made for remaining overgeneralization, which might make the plan either inconsistent or inefficient in use. Finally, the generalized plan, termed a MACROP, is stored away for future use.

In the example above, the generalized plan would be

	GOTODOOR (dx)
	GOTHRUDOOR (dx,rx1)
	GOTOB (bx)
with preconditions:	INROOM (ROBOT,rx2)
	CONNECTS (dx,rx2,rx1)
	STATUS (dx,OPEN)
	INROOM (bx,rx1)
	TYPE (bx,OBJECT)
and add list:	NEXTTO (ROBOT,bx)
	INROOM (ROBOT,rx1)

That is, the generalized plan sends the robot from any room through a connecting door to an object in the adjacent room.

Using the MACROP to guide execution. When STRIPS produces a detailed plan to achieve a goal, it does not necessarily follow that the robot should execute the plan exactly as given. One possibility is that some action fails to achieve its expected effect, so that the corresponding step of the plan needs to be repeated. Another is that the plan found is less than optimal and would be improved by omitting some steps entirely. The necessary flexibility during execution is provided by using the MACROP rather than the detailed plan in monitoring the robot's actions.

At the beginning of execution, the parameters of the MACROP are partially instantiated to the case at hand. The robot then attempts, at each stage, to execute the highest numbered step of the plan whose preconditions are satisfied. This procedure omits unnecessary steps and allows repeated execution, possibly with changed parameters, of a step that has failed. If there is no step whose preconditions are satisfied, replanning occurs. Determining which step can be done next is accomplished by a scan that exploits the design of the triangle table.

Using MACROPs in planning. When STRIPS is given a new problem, the time it takes to produce an answer can be reduced very considerably if there exists a MACROP that can be incorporated into its solution. The MACROP given above, for example, could be used as the first part of a plan to fetch a box from an adjacent room. The part of the MACROP consisting of its first two suboperators, if used alone, would also give a ready-made solution to the problem "Go to an adjacent room"; or it could be used repeatedly in solving "Go to a distant room."

The triangle table provides the means of determining whether a relevant macro operator exists. To determine whether the sequence of operators 1 through i of the MACROP is relevant, STRIPS checks the add list of this sequence as given by the i+1*th* row of the table. Once a MACROP is selected, irrelevant operators are edited out by a staightforward algorithm, leaving an economical, possibly parameterized operator for achieving the desired add list. The operator's preconditions are taken from the appropriate cells of column 0. Thus, almost any sub-sequence of operators from a MACROP can become a macro operator in a new plan. To keep new MACROPs from producing an overwhelming number of different operators that must be considered during planning, the system contains provisions for preventing consideration of redundant parts of overlapping MACROPs and for deleting MACROPs that have been completely subsumed by new ones.

In a sequence of problems given to STRIPS, the use of MACROPs in some cases reduced planning time by as much as two-thirds. The longest plan so formed, consisting of 11 primitive operations, took the robot from one room to a second room, opened a door leading to a third room, took the robot through the third room to a fourth room, and then pushed two pairs of boxes together. One drawback noted by the authors was that, however long the solution sequence, STRIPS at each stage of its search dealt with every operation in complete detail. A later program, Sacerdoti's ABSTRIPS (Article D6), provides the mechanism for deferring the details of the solution until after its main outline has been completed.

References

See Fikes & Nilsson (1971), Fikes, Hart, & Nilsson (1972), and Garvey & Kling (1969).

D6. ABSTRIPS

A combinatorial explosion faces all problem solvers that attempt to use heuristic search in a sufficiently complex problem domain. A technique called *hierarchical search* or *hierarchical planning*, implemented in Earl Sacerdoti's ABSTRIPS (1974), is an attempt to reduce the combinatorial problem. The idea is to use an approach to problem solving that can recognize the most significant features of a problem, develop an outline of a solution in terms of those features, and deal with the less important details of the problem only after the outline has proved adequate.

The implementation of this approach involves using distinct levels of problem representation. A simplified version of the problem, from which details have been omitted, occurs in a *higher level problem space* or *abstraction space*; the detailed version, in a *ground space*. By a slight extension, providing for several levels of detail instead of just two, a hierarchy of problem spaces is obtained. In general, each space in the hierarchy serves both as an abstraction space for the more detailed space just below it and as a ground space with respect to the less detailed space just above.

Background--The STRIPS System

ABSTRIPS is a modification of the STRIPS system, described in Article D5. The problem domain for both programs is a world of robot planning. In both, the program is given an initial state of the world, or *world model*, consisting of a set of formulas that describe the floor plan of a group of rooms and other facts such as the location of the robot and other objects within these rooms. The goal state to be achieved is also given. The elements of a solution sequence are *operators* representing robot actions; examples are operators for going up to an object, pushing an object, and going through a door. The definition of each operator contains three kinds of formulas: (a) its *preconditions*, representing statements that must be true of a world model in order for the operator to be applicable; (b) its *add list*, a list of formulas that will become true and should be added to the world model when the operator is applied; and (c) its *delete list*, a corresponding list of formulas to be deleted from the model upon application of the operator. The search for a sequence of operators producing the desired world model is guided by a *means-ends analysis* similar to that of GPS (Article D2).

Abstraction Spaces

Given the world models and operator descriptions of the basic STRIPS system, the first question is how to define the "details" that are to be ignored in the first pass at a solution. Sacerdoti's answer was to treat as details certain parts of the operator preconditions. At all levels of abstraction, the world models and the add and delete lists of operators remain exactly the same. Such a definition of "details" was found to be strong enough to produce real improvements in problem-solving efficiency, while keeping a desirable simplicity in the relationship between each abstraction space and its adjacent ground space.

The preconditions for an operator are stated as a list of predications, or *literals*, concerning the world model to which the operator is to be applied. The relative importance of literals is indicated by attaching to each a number called its *criticality value*. The hierarchy of problem spaces is then defined in terms of levels of criticality: In the space of criticality n, all operator preconditions with criticality less than n are ignored.

The assignment of criticality values is done just once for a given definition of the problem domain. The general ideas to be reflected in the assignment are the following:

1. If the truth value of a literal cannot be changed by any operator in the problem domain, it should have the highest criticality.

2. If the preconditions for an operator include a literal L that can be readily achieved once other preconditions for the same operator are satisfied, then L should be less critical than those other preconditions.

3. If the possibility of satisfying literal L depends on additional preconditions besides those referred to in (2), then L should have high but less than maximum criticality.

The actual assignment of criticalities is done by a combination of manual and automatic means. First, the programmer supplies a partial ordering of all predicates that can appear in operator preconditions. The partial ordering serves two purposes: It supplies a tentative criticality value for all instances of each predicate, and it governs the order in which the program will consider literals for possible increases (but not decreases) in criticality.

As an example, consider an operator TURN-ON-LAMP (x), with preconditions

TYPE (x,LAMP) and THERE EXISTS (r) [INROOM (ROBOT,r) and INROOM (x,r)) and PLUGGED-IN (x) and NEXTTO (ROBOT,x)] .

The partial ordering of predicates, reflecting an intuitive view of their relative importance, might be as follows:

Predicate	Rank
TYPE	4
INROOM	3
PLUGGED-IN	2
NEXTTO	1

Figure 1. Initial ranking of predicates.

The assignment algorithm, whose output is summarized in the figure below, would first find that the truth of TYPE (x,LAMP) is beyond the power of any operator to change and therefore would set its criticality to the maximum; in this case, 6. Then it would find that TYPE (x,LAMP) is an insufficient basis for achieving INROOM (ROBOT,r) or INROOM (x,r); so these two literals would have their criticality raised to the next highest value, 5. Next PLUGGED-IN (x) is considered, and a plan to achieve PLUGGED-IN (x) is found using only the literals already processed as a starting point. Hence, the PLUGGED-IN literal retains its tentative criticality of 2, and, similarly, NEXTTO (ROBOT,x) is given criticality 1. The result, after similar processing of the preconditions of the other operators in the domain, is a hierarchy of at least four, and possibly six, distinct problem spaces.

Literal	Criticality Value
TYPE (x,LAMP)	6
INROOM (ROBOT,r)	5
INROOM (x,r)	5
PLUGGED-IN (x)	2
NEXTTO (ROBOT,x)	1

Figure 2. Final criticality of literals.

Control Structure

A problem statement for ABSTRIPS, as for STRIPS, consists of a description of the state of the world to be achieved. A solution is a plan, or sequence of operators, for achieving it. ABSTRIPS proceeds by forming a crude plan at the highest level of abstraction and successively refining it. The executive is a recursive program taking two parameters: the current level of criticality, defining the abstraction space in which planning is to take place, and a list of nodes representing the plan to be refined. Before the initial call, criticality is set to the maximum, and the skeleton plan is initialized to a single operator--a dummy--whose preconditions are precisely the goal to be achieved. ABSTRIPS computes the difference between the preconditions and the current world model, finds operators relevant to reducing the difference and, if necessary, pursues subgoals of satisfying the preconditions of the selected operators. During this process, any preconditions of less than the current criticality are ignored. A search tree is built from which, if the process succeeds, a fuller operator sequence leading from the initial world model to the goal can be reconstructed. This new skeleton plan, together with the next lower criticality level, are passed recursively to the executive for the next round of planning.

The search strategy used by ABSTRIPS can be called *length-first*, in that the executive forms a complete plan for reaching the goal in each abstraction space before considering plans in any lower level space. This approach has the advantage that it permits early recognition of dead ends, thus reducing the work wasted in extending the search tree along fruitless paths involving detailed preconditions. If a subproblem in any particular space cannot be solved, control is returned to its abstraction space, and the search tree is restored to its previous state in that space. The node that caused the failure in the lower level space is eliminated from consideration and the search is continued in the higher level space. This mechanism, an example of *backtracking*, suffers from the problem that no information is available at the higher level on what caused the plan to fail.

Because backtracking can be inefficient and also because each operator in an abstraction space may be expanded to several operators in the ground space, it is important for ABSTRIPS to produce good plans at the highest level. Two modifications to STRIPS were made to try to insure that it would do so.

First, whereas a STRIPS search tended to be *depth-first* and therefore sometimes found non-optimal solutions, ABSTRIPS makes the order of expanding nodes in the search tree dependent on the level of abstraction. At the highest level it uses an *evaluation function* that may increase the search effort but which insures that the shortest possible solution sequence will be found. (See Article C3b on A*.)

The second change relates to the instantiation of operator parameters, in cases where two or more choices seem equally good. While STRIPS made a choice arbitrarily, ABSTRIPS defers the choice until a greater level of detail indicates one to be preferable. Backtracking can still occur should the choice be mistaken.

Performance

ABSTRIPS and STRIPS were compared on a sequence of problems. One of the longest, needing 11 operators for its solution, required the robot to open a door, go through the adjacent room to another room, push two boxes together, and then go through two more doors to reach the room where it was to stop. The basic STRIPS system required over thirty minutes of computer time to find the solution; ABSTRIPS used 5:28 minutes and generated only half the number of search-tree nodes. It was noted that by the time ABSTRIPS reached the most detailed level, it had in effect replaced the original large problem by a sequence of 7 easy subproblems.

References

See Sacerdoti (1974).

References

Adelson-Velskiy, G. M., Arlazarov, V. L., & Donskoy, M. V. Some methods of controlling the tree search in chess programs. **Artificial Intelligence**, 1975, 6, 361-371.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. **The Design and Analysis of Computer Algorithms**. Reading, Mass.: Addison-Wesley, 1974.

Amarel, S. On representations of problems of reasoning about actions. In D. Michie (Ed.), **Machine Intelligence 3**. New York: American Elsevier, 1968. Pp. 131-171.

Baudet, G. M. On the branching factor of the alpha-beta pruning algorithm. **Artificial Intelligence**, 1978, 10, 173-199.

Berliner, H. J. Some necessary conditions for a master chess program. In **IJCAI 3**, 1973. Pp. 77-85.

Berliner, H. J. Chess as problem solving: the development of a tactics analyzer. Dept. of Computer Science, Carnegie-Mellon University, 1974.

Berliner, H. J. Experiences in evaluation with BKG--a program that plays backgammon. In **IJCAI 5**, 1977. Pp. 428-433. (a)

Berliner, H. J. A representation and some mechanisms for a problem-solving chess program. In M. R. B. Clarke (Ed.), **Advances in Computer Chess 1**. Edinburgh: Edinburgh University Press, 1977. Pp. 7-29. (b)

Berliner, H. J. Search and knowledge. In **IJCAI 5**, 1977. Pp. 975-979. (c)

Berliner, H. J. A chronology of computer chess and its literature. **Artificial Intelligence**, 1978, 10, 201-214. (a)

Berliner, H. J. The B* search algorithm: a best-first proof procedure. CMU-CS-78-112, Dept. of Computer Science, Carnegie-Mellon University, 1978. (b)

Bernstein, A., Arbuckle, T., Roberts, M. de V., & Belsky, M. A. A chess playing program for the IBM 704. In **Proc. Western Joint Computer Conference**, 1958. New York: American Institute of Electrical Engineers, 1959. Pp. 157-159.

Bratko, I., Kopec, D., & Michie, D. Pattern-based representation of chess end-game knowledge. **Computer J.**, 1978, 21, 149-153.

Chang, C. L., & Slagle, J. R. An admissible and optimal algorithm for searching AND/OR graphs. **Artificial Intelligence**, 1971, 2, 117-128.

Charness, N. Human chess skill. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 34-53.

de Champeaux, D., & Sint, L. An improved bi-directional search algorithm. In **IJCAI 4**, 1975. Pp. 309-314.

de Champeaux, D., & Sint, L. An improved bi-directional heuristic search algorithm. **J. ACM**, 1977, **24**, 177-191.

Dijkstra, E. W. A note on two problems in connection with graphs. **Numerische Mathematik**, 1959, **1**, 269-271.

Doran, J. An approach to automatic problem-solving. In N. L. Collins & D. Michie (Eds.), **Machine Intelligence 1**. New York: American Elsevier, 1967. Pp. 105-123.

Doran, J. E., & Michie, D. Experiments with the graph traverser program. **Proceedings of the Royal Society of London**, 1966, **294** (series A), 235-259.

Elcock, E. W. Representation of knowledge in a geometry machine. In E. W. Elcock & D. Michie (Eds.), **Machine Intelligence 8**. New York: John Wiley & Sons, 1977. Pp. 11-29.

Ernst, G., & Newell, A. GPS: **A Case Study in Generality and Problem Solving**. New York: Academic Press, 1969.

Feigenbaum, E. A. Artificial intelligence: Themes in the second decade. In A. J. H. Morrell (Ed.), **Information Processing 68: Proc. IFIP Congress 1968 (Vol. 2)**. Amsterdam: North-Holland, 1969. Pp. 1008-1024.

Feigenbaum, E. A.,, & Feldman, J. (Eds.) **Computers and Thought**. New York: McGraw-Hill, 1963.

Fikes, R. E., Hart, P., & Nilsson, N. J. Learning and executing generalized robot plans. **Artificial Intelligence**, 1972, **3**, 251-288.

Fikes, R. E., & Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. **Artificial Intelligence**, 1971, **2**, 189-208.

Frey, P. W. An introduction to computer chess. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 54-81.

Fuller, S. H., Gaschnig, J. G., & Gillogly, J. J. Analysis of the alpha-beta pruning algorithm. Department of Computer Science, Carnegie-Mellon University, 1973.

Garvey, T., & Kling, R. User's guide to QA3.5 question-answering system. Technical Note 15, AI Group, Stanford Research Institute, Menlo Park, Calif., 1969.

Gaschnig, J. Exactly how good are heuristics? Toward a realistic predictive theory of best-first search. In **IJCAI 5**, 1977. Pp. 434-441.

Gelernter, H. A note on syntactic symmetry and the manipulation of formal systems by machine. **Information and Control**, 1959, **2**, 80-89.

Gelernter, H. Realization of a geometry-theorem proving machine. In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 134-152.

Gelernter, H., Hansen, J. R., & Gerberich, C. L. A Fortran-compiled list processing language. **J. ACM**, 1960, 7, 87-101.

Gelernter, H., Hansen, J. R., & Loveland, D. W. Empirical explorations of the geometry-theorem proving machine. In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 153-163.

Gelernter, H., & Rochester, N. Intelligent behavior in problem-solving machines. **IBM J. R&D**, 1958, 2, 336-345.

Gelperin, D. On the optimality of A*. **Artificial Intelligence**, 1977, 8, 69-76.

Gillogly, J. J. The technology chess program. **Artificial Intelligence**, 1972, 3, 145-163.

Gilmore, P. C. An examination of the geometry theorem machine. **Artificial Intelligence**, 1970, 2, 171-187.

Good, I. J. A five-year plan for automatic chess. In E. Dale & D. Michie (Eds.), **Machine Intelligence 2**. New York: American Elsevier, 1968. Pp. 89-118.

Greenblatt, R. D., Eastlake, D. E., & Crocker, S. D. The Greenblatt chess program. In **AFIPS Conference Proc., Fall Joint Computer Conference, 1967**. Washington, D. C.: Thompson Books, 1967. Pp. 801-810.

Griffith, A. K. A comparison and evaluation of three machine learning procedures as applied to the game of checkers. **Artificial Intelligence**, 1974, 5, 137-148.

Hall, P. A. V. Branch-and-bound and beyond. In IJCAI 2, 1971. Pp. 641-650.

Harris, L. R. The bandwidth heuristic search. In IJCAI 3, 1973. Pp. 23-29.

Harris, L. R. The heuristic search under conditions of error. **Artificial Intelligence**, 1974, 5, 217-234.

Harris, L. R. The heuristic search and the game of chess: a study of quiescence, sacrifices, and plan oriented play. In IJCAI 4, 1975. Pp. 334-339.

Harris, L. R. The heuristic search: an alternative to the alpha-beta minimax procedure. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 157-166.

Hart, P. E., Nilsson, N. J., & Raphael, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Trans. SSC**, 1968, SSC-4, 100-107.

Hart, P. E., Nilsson, N. J., & Raphael, B. Correction to *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. SIGART Newsletter, No. 37, December 1972, pp. 28-29.

Hearst, E. Man and machine: chess achievements and chess thinking. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 167-200.

Hillier, F. S., & Lieberman, G. J. **Operations Research** (2nd ed.). San Francisco: Holden-Day, 1974.

Jackson, P. C. **Introduction to Artificial Intelligence**. New York: Petrocelli, 1974.

Karp, R. M. Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), **Complexity of Computer Computations**. New York: Plenum Press, 1972. Pp. 85-103.

Kister, J., Stein, P., Ulam, S., Walden, W., & Wells, M. Experiments in chess. **J. ACM**, 1957, **4**, 174-177.

Knuth, D. E., & Moore, R. W. An analysis of alpha-beta pruning. **Artificial Intelligence**, 1975, **6**, 293-326.

Kotok, A. A chess playing program. RLE and MIT Computation Center Memo 41, Artificial Intelligence Project, Massachusetts Institute of Technology, 1962.

Kowalski, R. And-or graphs, theorem-proving graphs, and bi-directional search. In B. Meltzer & D. Michie (Eds.), **Machine Intelligence 7**. New York: John Wiley & Sons, 1972. Pp. 167-194.

Lawler, E. W., & Wood, D. E. Branch-and-bound methods: A survey. **Operations Research**, 1966, **14**, 699-719.

Levi, G., & Sirovich, F. A problem reduction model for non-independent subproblems. In **IJCAI 4**, 1975. Pp. 340-344.

Levi, G., & Sirovich, F. Generalized AND/OR graphs. **Artificial Intelligence**, 1976, **7**, 243-259.

Levy, D. The computer chess revolution. **Chess Life & Review**, February 1979, 84-85.

Manove, M., Bloom, S., & Engelman, E. Rational functions in MATHLAB. In D. G. Bobrow (Ed.), **Symbol Manipulation Languages and Techniques**. Amsterdam: North-Holland, 1968. Pp. 86-102.

Martelli, A. On the complexity of admissible search algorithms. **Artificial Intelligence**, 1977, **8**, 1-13.

Martelli, A., & Montanari, U. Additive AND/OR graphs. In **IJCAI 3**, 1973. Pp. 1-11.

Michie, D. Strategy building with the graph traverser. In N. L. Collins & D. Michie (Eds.), **Machine Intelligence 1**. New York: American Elsevier, 1967. Pp. 135-152.

Michie, D. A theory of advice. In E. W. Elcock & D. Michie (Eds.), **Machine Intelligence 8**. New York: John Wiley & Sons, 1977. Pp. 151-168.

Michie, D., & Bratko, I. Advice table representat[illegible]s of chess end-game knowledge. In **Proc. AISB/GI Conference on Artificial Intelligence**, 1978. Pp. 194-200.

Michie, D., & Ross, R. *Experiments with the adaptive graph traverser.* In B. Meltzer & D. Michie (Eds.), **Machine Intelligence 5**. New York: American Elsevier, 1970. Pp. 301-318.

Minsky, M. Steps toward artificial intelligence. In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 406-450.

Mittman, B. A brief history of the computer chess tournaments: 1970-1975. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 1-33.

Moore, E. F. The shortest path through a maze. In **Proceedings of an International Symposium on the Theory of Switching, Part II**. Cambridge: Harvard University Press, 1959. Pp. 285-292.

Moses, J. Symbolic integration. MAC-TR-47, Project MAC, Massachusetts Institute of Technology, 1967.

Newborn, M. **Computer Chess**. New York: Academic Press, 1975.

Newborn, M. The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores. **Artificial Intelligence**, 1977, 8, 137-153.

Newborn, M. Computers and chess news: recent tournaments. **SIGART Newsletter**, No. 65, April 1978, p. 11.

Newell, A., & Ernst, G. The search for generality. In W. A. Kalenich (Ed.), **Information Processing 1965: Proc. IFIP Congress 65**. Washington: Spartan Books, 1965. Pp. 17-24.

Newell, A., Shaw, J. C., & Simon, H. A. A variety of intelligent learning in a general problem-solver. In M. C. Yovits & S. Cameron (Eds.), **Self-organizing Systems**. New York: Pergamon Press, 1960. Pp. 153-189.

Newell, A., Shaw, J. C., & Simon, H. A. Chess-playing programs and the problem of complexity. *In E. A. Feigenbaum & J. Feldman* (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 39-70. (a)

Newell, A., Shaw, J. C., & Simon, H. A. Empirical explorations with the logic theory machine: A case history in heuristics. In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 109-133. (b)

Newell, A., & Simon, H. A. GPS, a program that simulates human tho[illegible] In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 279-293.

Newell, A., & Simon, H. A. **Human Problem Solving**. Englewood Cliffs, N. J.: Prentice-Hall, 1972.

Newell, A., & Simon, H. A. Computer science as empirical inquiry: Symbols and search. The 1976 ACM Turing Lecture. Comm. ACM, 1976, 19, 113-126.

Nilsson, N. J. Searching problem-solving and game-playing trees for minimal cost solutions. In A. J. H. Morrell (Ed.), Information Processing 68: Proc. IFIP Congress 1968 (Vol. 2). Amsterdam: North-Holland, 1969. Pp. 1556-1562.

Nilsson, N. J. Problem-Solving Methods in Artificial Intelligence. New York: McGraw-Hill, 1971.

Nilsson, N. J. Artificial intelligence. In J. L. Rosenfeld (Ed.), Information Processing 74: Proc. IFIP Congress 74. Amsterdam: North-Holland, 1974. Pp. 778-801.

Pitrat, J. A chess combination program which uses plans. Artificial Intelligence, 1977, 8, 275-321.

Pohl, I. Bi-directional and heuristic search in path problems. SLAC Report No. 104, Stanford Linear Accelerator Center, Stanford, 1969.

Pohl, I. First results on the effect of error in heuristic search. In B. Meltzer & D. Michie (Eds.), Machine Intelligence 5. New York: American Elsevier, 1970. Pp. 219-236. (a)

Pohl, I. Heuristic search viewed as path finding in a graph. Artificial Intelligence, 1970, 1, 193-204. (b)

Pohl, I. Bi-directional search. In B. Meltzer & D. Michie (Eds.), Machine Intelligence 6. New York: American Elsevier, 1971. Pp. 127-140.

Pohl, I. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In IJCAI 3, 1973. Pp. 12-17.

Pohl, I. Practical and theoretical considerations in heuristic search algorithms. In E. W. Elcock & D. Michie (Eds.), Machine Intelligence 8. New York: John Wiley & Sons, 1977. Pp. 55-72.

Polya, G. How to Solve It (2nd ed.). New York: Doubleday Anchor, 1957.

Raphael, B. The Thinking Computer. San Francisco: W. H. Freeman, 1976.

Reingold, E. M., Nievergelt, J., & Deo, N. Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, N. J.: Prentice-Hall, 1977.

Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. Artificial Intelligence, 1974, 5, 115-135.

Samuel, A. L. Some studies in machine learning using the game of checkers. In E. A. Feigenbaum & J. Feldman (Eds.), Computers and Thought. New York: McGraw-Hill, 1963. Pp. 71-105.

Samuel, A. L. Some studies in machine learning using the game of checkers. II--recent progress. **IBM J. R&D**, 1967, 11, 601-617.

Sandewall, E. J. Heuristic search: Concepts and methods. In N. V. Findler & B. Meltzer (Eds.), **Artificial Intelligence and Heuristic Programming**. New York: American Elsevier, 1971. Pp. 81-100.

Shannon, C. E. Programming a computer for playing chess. **Philosophical Magazine** (Series 7), 1950, 41, 256-275.

Shannon, C. E. A chess-playing machine. In J. R. Newman, **The World of Mathematics** (vol. 4). New York: Simon & Schuster, 1956. Pp. 2124-2133.

Simon, H. A., & Kadane, J. B. Optimal problem-solving search: All-or-none solutions. **Artificial Intelligence**, 1975, 6, 235-247.

Slagle, J. R. A heuristic program that solves symbolic integration problems in freshman calculus: Symbolic Automatic Integrator (SAINT). 5G-0001, Lincoln Laboratory, Massachusetts Institute of Technology, 1961.

Slagle, J. R. A heuristic program that solves symbolic integration problems in freshman calculus. In E. A. Feigenbaum & J. Feldman (Eds.), **Computers and Thought**. New York: McGraw-Hill, 1963. Pp. 191-203. (Also in **J. ACM**, 1963, 10, 507-520.)

Slagle, J. R. **Artificial Intelligence: The Heuristic Programming Approach**. New York: McGraw-Hill, 1971.

Slagle, J. R., & Dixon, J. K. Experiments with some programs that search game trees. **J. ACM**, 1969, 16, 189-207.

Slagle, J. R., & Dixon, J. K. Experiments with the M & N tree-searching program. **Comm. ACM**, 1970, 13, 147-154.

Slate, D. J., & Atkin, L. R. CHESS 4.5--the Northwestern University chess program. In P. W. Frey (Ed.), **Chess Skill in Man and Machine**. New York: Springer-Verlag, 1977. Pp. 82-118.

Thorp, E., & Walden, W. E. A computer-assisted study of Go on m x n boards. In R. B. Banerji & M. D. Mesarovic (Eds.), **Theoretical Approaches to Non-Numerical Problem Solving**. Berlin: Springer-Verlag, 1970. Pp. 303-343.

Turing, A. M., et al. Digital computers applied to games. In B. V. Bowden (Ed.), **Faster Than Thought**. London: Pitman, 1953. Pp. 286-310.

Vanderbrug, G., & Minker, J. State-space, problem-reduction, and theorem proving--some relationships. **Comm. ACM**, 1975, 18, 107-115.

Whitehead, A. N., & Russell, B. **Principia Mathematica** (2nd ed., Vol. 1). Cambridge: The University Press, 1925.

Wilkins, D. Using plans in chess. To appear in **IJCAI 6**, 1979.

Winston, P. H. **Artificial Intelligence.** Reading, Mass.: Addison-Wesley, 1977.

Index